ESC-EN-HA-TR-2012-109

Technical Report
1166

Survey of Cyber Moving Targets

H. Okhravi
M.A. Rabe
T.J. Mayberry
W.G. Leonard
T.R. Hobson
D. Bigelow
W.W. Streilein

25 September 2013

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

Prepared for the Department of Defense under Air Force Contract FA8721-05-C-0002.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **25 SEP 2013**	2. REPORT TYPE **N/A**	3. DATES COVERED

4. TITLE AND SUBTITLE **Survey of Cyber Moving Target Techniques**	5a. CONTRACT NUMBER **FA8721-05-C-0002**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) **H. Okhravi /M.A. Rabe, T.J. Mayberry, W.G. Leonard, T.R. Hobson, D. Bigelow, and W.W. Streilein**	5d. PROJECT NUMBER **2084**
	5e. TASK NUMBER **273**
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **MIT Lincoln Laboratory 244 Wood Street Lexington MA 02420-9108**	8. PERFORMING ORGANIZATION REPORT NUMBER **TR-1166**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) **DoD**	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This survey provides an overview of different cyber moving target techniques, their threat models, and their technical details. A cyber moving target technique refers to any technique that attempts to defend a system and increase the complexity of cyber attacks by making the system less homogeneous, less static, and less deterministic. In this survey, we describe the technical details of each technique, identify the proper threat model associated with the technique, and identify its implementation and operational cost. Moreover, we describe the weaknesses of each technique based on the current proposed attacks and bypassing exploits, and provide possible directions for future research in that area.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**149**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

This report is based on studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This work was sponsored by the Department of Defense, under Air Force Contract FA8721-05-C-0002.

Massachusetts Institute of Technology
Lincoln Laboratory

Survey of Cyber Moving Targets

H. Okhravi
M.A. Rabe
W.G. Leonard
T.R. Hobson
D. Bigelow
W.W. Streilein
Group 58

T.J. Mayberry
formerly Group 58

Technical Report 1166

25 September 2013

Lexington Massachusetts

EXECUTIVE SUMMARY

This survey provides an overview of different cyber moving target techniques, their threat models, and their technical details. A cyber moving target technique refers to any technique that attempts to defend a system and increase the complexity of cyber attacks by making the system less homogeneous, less static, and less deterministic. In this survey, we describe the technical details of each technique, identify the proper threat model associated with the technique, and identify its implementation and operational cost. Moreover, we describe the weaknesses of each technique based on the current proposed attacks and bypassing exploits, and provide possible directions for future research in that area.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

1. INTRODUCTION AND TAXONOMY

This survey provides an overview of different cyber moving target techniques, their threat models, and their technical details. A cyber moving target technique refers to any technique that attempts to defend a system and increase the complexity of cyber attacks by making the system less homogeneous, less static, and less deterministic [1]. In this survey, we describe the technical details of each technique, identify the proper threat model associated with the technique, and identify its implementation and operational cost. Moreover, we describe the weaknesses of each technique based on the current proposed attacks and bypassing exploits, and provide possible directions for future research in that area.

1.1 TAXONOMY OF MOVING TARGET TECHNIQUES

We could identify five top-level categories and two subcategories for moving target techniques. Here we give a short description for each category.

1. **Dynamic Runtime Environment:** Techniques that change the environment presented to an application by the operating system (OS) during execution dynamically.

   1.1. **Address Space Randomization:** Techniques that change the layout of memory dynamically. This can include the location of program code, libraries, stack/heap, and individual functions.

   1.2. **Instruction Set Randomization:** Techniques that change the interface presented to an application by the OS dynamically [58]. The interface can include the processor and system calls used to manipulate the input/output (I/O) devices.

2. **Dynamic Software:** Techniques that change application's code dynamically. The change can include modifying the program instructions, their order, their grouping, and their format.

3. **Dynamic Data:** Techniques that change the format, syntax, encoding, or representation of application data dynamically.

4. **Dynamic Platforms:** Techniques that change platform properties (e.g., central processing unit (CPU), OS) dynamically. This can include the OS version, CPU architecture, OS instance, platform data format, etc.

5. **Dynamic Networks:** Techniques that change network properties including protocols or addresses dynamically.

1.2 TAXONOMY OF ATTACK TECHNIQUES

The effect of each moving target technique is described in terms of the attack technique that it mitigates. Here we provide a brief definition for the attack techniques used in this report. The taxonomy of attacks is a customized version of the Common Attack Pattern Enumeration and Classification (CAPEC) attack categories [106].

1. **Data Leakage Attacks:** Attacks that actively target important information on a system, e.g., leakage of crypto keys from memory.
2. **Resource Attacks:** Attacks that exhaust or manipulate shared resources in a system, e.g., denial-of-service (DoS) using CPU saturation.
3. **Injection:** Attacks that force malicious behavior in the system.
   - 3.1. **Code Injection:** Attacks that force malicious behavior in the system by inserting malicious code, e.g., buffer overflow and script injection, and Structured Query Language (SQL) injection.
   - 3.2. **Control Injection:** Attacks that force malicious behavior in the system by manipulating the control of the system and without malicious code. Control can include timing, ordering, and arguments of different operations, e.g., chaining existing code snippets together to achieve malicious behavior—return-oriented programming (ROP) [81].
4. **Spoofing:** Attacks that fake identity of a user or a system, e.g., man-in-the-middle attack and phishing attack.
5. **Exploitation of Authentication:** Attacks that compromise explicit or implicit authentication processes in a system, e.g., cross-site scripting (XSS).
6. **Exploitation of Privilege/Trust:** Attacks that misuse granted privileges, e.g., session hijacking.
7. **Scanning:** Attacks that collect information passively or nonintrusively, e.g., port scanning.
8. **Supply Chain/Physical Attacks:** Attacks that target supply chain or physical security of a system, e.g., malicious processor.

1.3 TAXONOMY OF ENTITIES PROTECTED

Each moving target technique is designed to protect specific entities in a system. Here we provide a taxonomy of entities protected by the techniques we analyze in this survey.

1. **Applications:** All or specific applications are protected from network entities or other applications running on the same system, e.g., protecting application memory location from other applications and protecting database applications.
2. **Operating System:** The operating system is protected from network entities or malicious applications running on top of it. This protection usually attempt to prevent privilege escalation or access to the kernel-space and other applications, e.g., sandboxing suspicious applications.

3. **Machine:** All or specific types of machines (also called clients, hosts, or servers) are protected from other network entities, e.g., changing the Internet Protocol (IP) addresses to make scanning more difficult and protecting web servers behind a firewall.

4. **Network:** A network or subnet is protected from other networks, e.g., dynamically changing IP address on the virtual private network (VPN) gateway to protect against malicious connections.

5. **Traffic:** Confidentiality and/or integrity of all or specific types of network traffic is protected, e.g., dynamically changing protocols to make traffic injection more difficult.

6. **Session:** A set of user operations (a session or a transaction) is protected from other untrusted operations, e.g., a secure web transaction is protected from other web pages browsed on the same machine.

7. **Data:** Confidentiality or integrity of data handled by applications or stored on the machine is protected, e.g., changing data encoding to prevent malicious data modifications.

1.4 CYBER KILL CHAIN

Each moving target technique is focused on disrupting certain phases of a successful attack. For instance, while a technique may make it less likely for an exploit to succeed during launch, another focuses on making information collection on the target more challenging. In this survey, we try to identify the phase of an attack each technique is targeting. These phases are also referred to as the cyber kill chain.

1. **Reconnaissance:** The attacker collects useful information about the target.

2. **Access:** The attacker tries to connect or communicate with the target to identify its properties (versions, vulnerabilities, configurations, etc.).

3. **Exploit Development:** The attacker develops an exploit for a vulnerability in the system in order to gain a foothold or escalate his privilege.

4. **Attack Launch:** The attacker delivers the exploit to the target. This can be through a network connection, using phishing-like attacks, or using a more sophisticated supply chain or gap jumping attack (e.g., infected USB drive).

5. **Persistence:** The attacker installs additional backdoors or access channels to keep his persistence and access to the system.

We choose this kill chain because it is well suited for the types of protections offered in moving target defenses. There are other types of kill chains proposed in the literature that are better suited for specific domains in cyber. They include *cyber war kill chain* (phases: reconnaissance, weaponize, delivery, exploit, install, command and control, and act on objectives), *action-oriented kill chain* (phases:

deter, protect, detect, react, and survive), *detection kill chain* (phases: herd, perturb, disturb, etc.), and others.

1.5 TAXONOMY OF WEAKNESSES

When identifying the weaknesses associated with each moving target technique, we consider four types of weaknesses that can make the technique ineffective. One or all of these weaknesses can exist in a technique and any of them can defeat the purpose of that technique.

1. **Overcome Movement:** With this weakness, the movement happens and the pattern of movement is random or controlled, but the adversary can still attack the surface protected by the moving target technique. For example, injecting many copies of the exploit to overcome address space randomization is a form of overcoming the movement.

2. **Predict Movement:** With this weakness, the movement happens and the pattern of movement is random or controlled, but the adversary can still attack the surface protected by the moving target technique. For example, leaking addresses to predict the location of libraries is a form of predicting the movement in address space randomization.

3. **Limit Movement:** With this weakness, the movement happens, but the pattern of movement is limited by adversary's actions. For example, the adversary can fill up memory to limit the randomness in address space randomization (a.k.a code spraying).

4. **Disable Movement:** With this weakness, the adversary explicitly disables the movement. For example, address space randomization can be disabled in the OS by pushing a bad configuration.

1.6 SCOPE

This survey tries to provide a complete *representative* set of moving target techniques from open and public sources of information. Although we expect that there are other commercial product or academic projects with different names that implement similar moving target techniques or some combination thereof, to the best of our knowledge they are not fundamentally different in their concepts and workings.

2. DYNAMIC RUNTIME ENVIRONMENT

2.1 ADDRESS SPACE RANDOMIZATION

2.1.1 Address Space Layout Permutation

Last Updated: 7/18/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Address Space Randomization

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [2] defends against buffer overflow attacks on the stack and heap from an adversary that can provide arbitrary input to a vulnerable program. A buffer overflow attack occurs when an attacker can provide malformed input to a program that causes it to write the input incorrectly to areas outside the allotted memory location. This technique defends against direct overflow attacks, where the goal is to overwrite the return pointer on the stack, and indirect attacks where the goal is to overwrite a function pointer on the heap that is later dereferenced. It does not protect against adversaries that have local access to a machine.

Description:

Details: This technique performs stack randomization at both the user and kernel levels. User-level permutation includes both a coarse randomization (code and data segments are randomly placed) and a fine-grained randomization (functions and variables are randomized inside code and data segments). The user-level permutation is implemented as a binary rewriting tool that processes Executable and Linkable Format (ELF) executables and outputs a randomized version with the same behavior. This rewriting does not require source code access or recompilation. At the kernel level, the starting location of the user stack is randomly chosen and the heap is removed from its usual place inside the data section and randomly placed in program memory. Additionally, the mmap() function is patched so that individual pages inside the heap are randomly allocated.

Entities Protected: All programs running on the machine are protected from code or control injection through individual, independent program randomization.

Deployment: This technique could be deployed on any generic machine.

Execution Overhead:

- The required kernel changes do not affect performance to a significant degree and user-level changes occur as a preprocessing step and so do not affect execution speed.

Memory Overhead:

- Experimental results show an approximately 20% increase in executable size and memory footprint.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: Memory randomization is more effective when it is combined with various types of memory guards [96–105].

Weaknesses: As with many other address randomization techniques, the entropy of this scheme is limited [78, 87] by the architecture machine width (i.e., number of bits: 32 or 64). In this case, they do get very close to that limit with 29 bits of entropy for the heap location, 28 bits for the stack location, 20 bits in mmap(), and 20 bits within the data and code segments. This far exceeds other related schemes. However, their scheme is not resistant to attacks that can violate "memory secrecy" [83] through leakage or local access. It cannot randomize inside of stack frames so it is also vulnerable to ROP attacks. It may also be vulnerable to a heap spraying technique [79] where large chunks are allocated quickly to try to reduce uncertainty on the heap.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: Attacker level of effort is raised substantially. Although it may be possible to mount attacks using address leakage, it would require additional effort that is much higher than finding and exploiting the original buffer overflow.

Availability: Prototyped by authors but not publicly available.

Additional Considerations: None

Proposed Research: Developing a memory protection technique that does not assume memory secrecy and provide high entropy is an important missing piece.

Funding: Unknown

2.1.2 DieHard

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Address Space Randomization

Threat Model:

Attack Technique Mitigated: Code and Control Injection

Details: DieHard [3, 85] protects the heap from indirect buffer overflow attacks where an attacker attempts to overwrite a function pointer to cause control injection.

Description:

Details: DieHard attempts to defend against four classes of vulnerabilities that could lead to program crash or code/control injection: invalid frees, buffer overflows, dangling pointers, and uninitialized reads. An invalid free is one where the program attempts a free operation on a pointer that has already been freed or on an object that was not dynamically allocated. A buffer overflow occurs when a program attempts to write to a location past the end of a buffer and instead overwrites a data location belonging to another object. A dangling pointer bug is present when an object is freed but pointers to it still remain and are eventually dereferenced. Uninitialized reads occur when a variable is declared and read before it is initialized. This usually results in the data that was previously in the location this variable was allocated at being read.

The strategy used has three main elements: address randomization, heap spacing, and replication. Addresses of heap objects are randomized using a different seed each time the program is executed. Additionally, the heap is sized to be *M* times larger than is necessary for program execution. This allows for extra space between objects so it is less likely that a buffer overflow will result in overwriting of another object. DieHard also maintains *N* copies of the heap initialized with different random seeds. Whenever a memory operation is done, a "vote" occurs between the copies. These three techniques together provide a probabilistic measure of defense against the four classes of vulnerabilities. Since there are multiple copies with different randomized addresses, any targeted buffer overflow would end up segmenting the control flow (i.e. replicas would end up executing different segments of code). This would be discovered and a recovery mechanism could possibly be used [89, 91, 93].

Entities Protected: Can be configured to protect any or all programs on a machine.

Deployment: This technique could be deployed on any generic machine by patching the OS.

Execution Overhead:

- Experimental results show an execution overhead of 50–100% with M = 2 and 3 replicas.

Memory Overhead:

- Because of the increased heap size and replicas the memory overhead is quite large, at least $M*N$.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☒ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: DieHard and address space layout randomization (ASLR) can interfere with each other and potentially have negative impact. DieHard consumes a large amount of memory, which makes the ASLR less effective.

Weaknesses: Provides only probabilistic security, depending on the parameters chosen a system might be vulnerable to a brute force attack. It also assumes "memory secrecy."

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Makes it difficult to mount injection attacks against the heap and even more difficult to ensure that a given attack will work 100% of the time.

Availability: The code is available online as well as a demonstration that is configured to provide heap randomization to Mozilla Firefox in Windows.

Additional Considerations:

- In the process of stopping buffer overflow attacks, this technique also allows programs to recover from many common errors without crashing (see [60] for failure oblivious computing). Most other memory randomization techniques will prevent an attacker from gaining control, but will still cause the program to crash upon attempted exploitation of a buffer overflow.

Proposed Research: A low-overhead memory protection technique that does not assume memory secrecy is still an open research problem. The memory overhead of DieHard is really significant.

Funding: National Science Foundation, Intel Corporation, Microsoft Research

2.1.3 Instruction Level Memory Randomization

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Address Space Randomization

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [4] defends against buffer overflow attacks on the stack and heap from an adversary that can provide arbitrary input to a vulnerable program. A buffer overflow attack occurs when an attacker can provide malformed input to a program that incorrectly causes it to write the input to areas outside the allotted memory location. This technique defends against direct overflow attacks, where the goal is to overwrite the return pointer on the stack, and indirect attacks where the goal is to overwrite a function pointer on the heap that is later dereferenced. It does not protect against adversaries that have local access to a machine.

Description:

Details: This technique randomizes both the stack and heap. The randomization takes the form of a program that transforms an executable into a randomized version that has the same behavior. Random padding is added at the start of the stack and before the return address in every stack frame by modifying the assembly code that creates these stack frames. The placement of heap chunks is also randomized by requesting a chunk much larger than is needed and then placing the original chunk randomly inside that larger chunk. The main advantage of this technique is that it does not need access to source code or recompilation of target programs. It matches with the current software distribution model in that it could be hooked into an installer application that would randomize the executable differently for every machine where it is deployed.

Entities Protected: Any or all programs running on a machine that have been processed by the binary rewriter.

Deployment: Can be deployed to any generic machine as part of a platform configuration or individual programs can be manually randomized. This method is a separate application and does not require modification to any other component.

Execution Overhead:

- None

Memory Overhead:

- Stack and heap size increased by approximately 20%.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This technique can be complimentary to ASLR, but it can have a conflict with DieHard. Applying both DieHard and this technique can make the memory overhead very large.

Weaknesses: This scheme only partially protects against return-oriented programming. It makes it more difficult to put arguments onto the stack that will be passed to the target library function, but does not fully prevent redirection of program control. The randomness injected is also limited by the machine architecture, namely it cannot be more than 32-bits (and it probably much lower than that in practice). They also cannot rewrite some instructions so in their experimental results they only protected about 70% of each executable. This technique is not effective against attacks that violate memory secrecy and may be vulnerable to heap spraying.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Increases the level of effort for attackers in many circumstances. Since some instruction sequences cannot be processed by this technique, portions of executables may remain vulnerable.

Availability: No code publicly available.

Additional Considerations:

- This approach is notably different from other memory randomization techniques in that it is done as a binary rewriting. This means that it could actually be installed on a software distribution server that would uniquely randomize executables as they were being distributed (and thus require no configuration or changes of any kind on the client).

Proposed Research: A low-overhead memory protection technique that does not assume memory secrecy is still an open research problem.

Funding: National Science Foundation

2.1.4 Operating System Randomization

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Address Space Randomization

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [5] attempts to defend against buffer overflow attacks through stack randomization as well as decrease the likelihood of injected code successfully running through library and system call randomization. It protects against an adversary that can control the input to an application or service.

Description:

Details: This technique was one of the earliest memory randomization attempts. The authors use three different techniques to add randomness to the program environment: stack randomization, system call randomization, and movement of libc. The starting location of the stack is randomly offset by a 15-bit value and the system call table is increased to 512 (9-bits). The starting location of libc is moved, but it is done semi-deterministically. This adds some heterogeneity to systems but is not hard to bypass.

Entities Protected: All programs running on a machine using this technique.

Deployment: This technique could be applied to any generic machine by modifying the OS.

Execution Overhead:

- None

Memory Overhead:

- Stack size is increased slightly due to the offset but it is negligible.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: ASLR has conflict with DieHard. The large amount of memory used in DieHard makes ASLR less effective. ASLR is more effective if it is combined with various memory guards [96–105].

Weaknesses: This scheme overall is very weak. Its redeeming quality is that it has essentially no runtime penalties and integrates seamlessly with the system. The amount of entropy it introduces is very easy to brute force and the system call randomization is not effective against return-oriented attacks that use library calls. It also does not prevent attacks using relative addresses [80, 84, 86, 88] as it only moves the starting location of the stack. The library protection itself can be circumvented as long as the attacker knows the system is using this approach (it makes no real attempt to randomize the library locations, only move them somewhere that an attacker is less likely to look).

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: Raises the level of effort required to exploit a buffer overflow attack and makes it harder to have a single attack that works on all vulnerable machines.

Availability: No source code is publicly available. Similar variants of ASLR is implemented in Windows Vista and 7 [6, 73, 76, 77], Mac OS X v10.5 [7] and newer, Linux since 2.6.12 [8], and iOS 4.3 and newer [9].

Additional Considerations: ASLR usually does not have a significant overhead, and once implemented, it can be applied easily to any system by patching the system. As a result, even considering its weaknesses it is advisable to use ASLR because there is no significant downside to it. ASLR-like techniques can also be implemented in the hardware (see [72]). For a formal model of ASLR-like defenses see [75].

Proposed Research: ASLR implementations suffer from a common set of problems that include low entropy, memory secrecy assumption, and limited application of randomization. An effective memory protection scheme must be developed that does not make these assumptions.

Funding: Unknown

2.1.5 Function Pointer Encryption

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Address Space Randomization

Threat Model:

Attack Technique Mitigated: Code Injection and Control Injection

Details: This technique [10] defends against control injection through indirect buffer overflow attacks on the heap by encrypting all function pointers so they cannot be modified.

Description:

Details: This technique aims to prevent indirect buffer overflow attacks by making it difficult for the attacker to overwrite a function pointer with a chosen value. The GNU Compiler Collection (GCC) is patched so that at link/load time all function pointers **fp* are replaced by **fp* XOR *address(fp)* XOR *rand* where *rand* is a 32-bit random number, chosen at the start of execution. Using *rand* provides a high degree of unpredictability if the attacker does not know it, and it is chosen independently at the start of every execution so it should be difficult to guess. Incorporating *address(fp)* makes two different pointers to the same function have different keys. Additionally, this makes it so that the attacker cannot learn an encrypted value for one pointer and substitute it for another, changing the location of the original pointer. The "key" is effectively *address(fp)* XOR *rand* and is used symmetrically to decrypt the respective function pointer when it is dereferenced. If an attacker manages to find a buffer overflow vulnerability and exploit it to overwrite a function pointer, he will not be able to forge an encrypted address that will point to his chosen location when it is decrypted (since he does not know *rand*).

Entities Protected: All programs running on a machine utilizing this technique.

Deployment: This technique could be applied to any generic machine by modifying the compiler and OS.

Execution Overhead:

- The authors show an experimental slowdown of approximately 4%.

Memory Overhead:

- The size of the executable in memory is increased by addition of the encryption/decryption keys. The paper does not measure this effect but it is likely small (each function pointer approximately doubles in size).

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: None

Weaknesses: This technique is vulnerable to an attacker that has a copy of the program and can learn the encrypted value of a function pointer at runtime (through violation of memory secrecy). The function pointer is masked by its own address, which can be determined by an attacker running a copy of the program, and a random key, which can be deduced if the encrypted function pointer is known along with the unencrypted function pointer and its address (since the encryption function is just XOR). This effectively recovers the secret encryption key and would allow an attacker to forge a pointer to any chosen location that would work for *any* function pointer in the program (not just the one that the attacker originally learned). Techniques exist that would allow an attacker to exploit a vulnerable program to obtain one or more encrypted function pointers.

The above threat can be partially mitigated by using a cryptographic hash function instead of *XOR* when combining *rand* and *addr(fp)*. This would still allow an attacker to forge the specific function pointer that was leaked to him, but it would not make other unrelated function pointers vulnerable (since the hash cannot be reversed and *rand* is not learned). Load time would be significantly slower while the linker computes hashes for each function pointer, but runtime would be the same because encryption and decryption would still be XOR (just the calculation of the individual keys changes). Full mitigation of this threat requires use of an encryption function that is secure against a known plaintext/ciphertext attack.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Makes it difficult for an attacker to redirect program control to a chosen address unless he can both obtain a copy of the program executable and violate memory secrecy to obtain encrypted addresses.

Availability: This technique is used in several Linux distributions that we know about. Fedora Core encrypts function pointers in libc but not in other programs or libraries. Red Hat Enterprise has a reference to encrypting function pointers in one of its whitepapers but it is unclear what the scope of it is in their implementation.

Additional Considerations: This technique is also like ASLR. It has no significant downside, so if it is available, it is advisable to use it even if it has weaknesses.

Proposed Research: In the original paper, XOR was chosen as an encryption function because it is very fast and causes little overhead in the program execution. Using a secure encryption function at the time was not possible. Recently, however, Intel has added a hardware instruction set for Advanced Encryption Standard (AES) that can encrypt/decrypt in a small number of cycles. We propose that this scheme be implemented with XOR replaced by AES encryption/decryption done in hardware in order to evaluate the effect on performance. Such a scheme would be secure against an attacker with any knowledge of the program except the encryption key. The question of where to store the encryption key is still open, but it should be possible to store it such that it would require an additional exploit in the kernel to bypass. It may also be possible to extend this technique to direct buffer overflow attacks (overwriting stack return addresses) but the implementation would be considerably different.

Barring AES encryption, this technique could also be made more robust by combining it with some kind of memory randomization. The most straightforward method would be to choose one that is implemented as a binary rewriter; from the point of view of the loader which does the encryption it would be no different but the executable on each machine would be randomized differently, making it much more difficult for an attacker (see above attack requirements).

Funding: National Science Foundation, Air Force Research Laboratory

2.2 INSTRUCTION SET RANDOMIZATION

2.2.1 G-Free

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Instruction Set Randomization (ISR)

Threat Model:

Attack Technique Mitigated: Control Injection

Details: This technique [11] aims to mitigate ROP attacks against executables compiled with the modified compiler. It does not fully protect against return-to-libc type attacks where the attacker wishes to execute an entire function from the target program.

Description:

Details: ROP attacks consist of an attacker redirecting control of a program back into itself at specific useful sequences of instructions. This way, no code needs to be injected but the attacker can still achieve malicious behavior by running pieces of the original executable in the wrong order to achieve arbitrary results. The authors note that all ROP attacks chain together pieces of code that ultimately each end with a *free branch* instruction. These free branch instructions are specific uses of return or jump instructions [82] where the target of the branch is dependent on a value on the stack or in a register (things that can be compromised by the attacker). If the attacker cannot find any useful code ending in a free branch, then he can only execute full function calls like in a return-to-libc attack, effectively eliminating generalized ROP.

The first step to stopping ROP is eliminating all misaligned free branch instructions. Since modern instruction sets are variable length, an attacker can often take a series of instructions and, by jumping into the middle of one of those instructions, execute an instruction on the CPU that never originally existed in the executable. This new instruction is a combination of the ending bits from one instruction and the starting bits of the next. Any free branch instructions that could be created in this way are a side effect of instruction ordering, and removing them would reduce the number of free branches available to an attacker by a large amount. They must be removed carefully, however, since the program semantics must remain unchanged. The authors accomplish this by scanning for these misaligned free branch instructions and inserting No Operation Performed (NOP) instructions to break them up. NOPs do not effect program execution and so can safely act as buffers to prevent adjoining instructions from incidentally creating a misaligned free branch. Additionally, these NOPs are arranged into a so-called *alignment sled*, which is a long sequence of NOPs, so that no matter what the alignment was when execution reached the start of the sled, by the time it reaches the end it will be realigned correctly. This is possible because NOPs are the shortest instruction and eventually execution will align onto one of them and continue normally.

The second protection mechanism used is a careful encryption of the return pointer on the stack. At the function call entry point, the return pointer is encrypted (using XOR with a random key) and pushed onto the stack. A set of instructions is also inserted as a footer, directly above the return instruction, so that the pointer is decrypted before return is called. If, at any point in the middle of the function, a stack overflow occurs, an attacker could not put a value into the return pointer that would be successfully decrypted into his target address. These two techniques together prevent generalized return-oriented attacks.

Entities Protected: Protects all binaries compiled with the modified compiler.

Deployment: Can be deployed on any generic machine by modifying the compiler.

Execution Overhead:

- Approximately 3% slowdown.

Memory Overhead:

- Approximately 26% increase in executable size.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: A ROP protection technique (such as G-Free) should ideally be combined with other memory protection techniques (such as ASLR or function pointer encryption).

Weaknesses: The encryption used is simply XOR so this technique relies on the fact that the attacker cannot read portions of the memory (memory secrecy) [56, 65, 74]. If the attacker could gain access to the return pointer value, he could recover the key and forge a new return pointer that would be interpreted correctly by the return instruction.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Restricts attackers to return-to-libc style attacks where whole functions are used instead of attacks using gadgets or misaligned instructions.

Availability: No code publicly available.

Additional Considerations: A ROP protection technique is only effective when it is applied to every application running on a machine. If an application or library is not compiled with this technique, the entire system is vulnerable to ROP attack. This makes compiler-level defenses against ROP limited in scope. There are similar techniques for protection against specific types of attacks [49] (e.g., spraying attacks).

Proposed Research: An OS-level protection against ROP is necessary to defend against ROP in all the libraries and applications. More importantly, the actual capability of ROP attacks is unknown at this point. More research is required to understand the full power of ROP attacks.

Funding: European Union Seventh Framework Programme and European Commission

2.2.2 Practical Software Dynamic Translation

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: Instruction Set Randomization

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This technique [12] protects against code injection into running binaries from all vectors. It does not protect against return oriented attacks and assumes that the OS is secure.

Description:

Details: Previous ISR techniques have two downsides that make them very unappealing: slow execution due to the requirement for an emulator to run any executable code and a weak encryption function, namely XOR. This scheme fixes the first problem by using a very lightweight virtual machine (VM) for execution and the second by switching to AES for encryption. They use an existing VM called Strata [48] for their ISR scheme, modified to allow for the necessary binary rewriting. When an executable is loaded from disk, Strata encrypts it block by block using AES. During execution, each time the program counter would point to an encrypted instruction, Strata decrypts the block that it is part of and continues by calling the regular fetch instruction. Each instruction also comes with a tag that can be verified so that after decryption Strata can decide whether the code is legitimate or if it has been injected. Any injected code could not match the tag, let alone produce a valid, useful instruction for the attacker since he does not know the encryption key used. To speed up execution, once the blocks are decrypted they are kept in a cache for reuse. The encryption key is generated fresh for every program execution and is kept by the VM so it cannot be read or altered by the program.

Entities Protected: All executables running on the Strata VM.

Deployment: Can be deployed on any generic machine by adding an extra virtualization layer.

Execution Overhead:

- Up to a 20% slowdown in execution.

Memory Overhead:

- Up to a 70% increase in executable size overhead and memory footprint.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: Because of relying on a virtualization layer, this technique is really a stand alone technique that does not combine well with other OS-level defenses.

Weaknesses: AES is used in Electronic Codebook (ECB) mode that encrypts two identical blocks to the same value. This means that an attacker could execute a replay attack by finding useful encrypted instructions that exist in the executable and injecting them as shellcode. ECB is used for efficiency reasons so that fewer decryptions are required. Additionally, the Strata VM itself becomes a new point of attack since it holds all the keys and is in charge of readying instructions for execution.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: This techniques makes it difficult for an attacker to inject arbitrary code into an executable. The attacker would need to brute force the encryption key in order to forge instructions that would decrypt to anything useful. More likely, the attack vector will shift to ROP, which is not mitigated with this technique.

Availability: No code publicly available.

Additional Considerations: The memory and execution overhead may become significant if all applications are virtualized in this manner. A similar technique is proposed in [55, 62].

Proposed Research: As with other techniques using encryption, this could benefit from hardware AES instructions recently added to Intel processors.

Funding: DARPA, National Science Foundation

2.2.3 RandSys

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategories: Address Space Randomization and ISR

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [13] defends against code injection and control injection from buffer overflow attacks on the stack and heap. This method is only focused on remote machine-code injection attacks. This method also assumes that the kernel is safe and it would not protect against kernel-level code injection attacks.

Description:

Details: This is a hybrid ISR and ASLR technique. It uses subsets of techniques from each category along with some additional guards to create a new implementation.

For ISR, it implements system call randomization between user space and kernel space (similar to [64]). When a process is created, the exec system call is intercepted in the kernel and control is given to RandSys. RandSys searches for all system calls in the application then takes their location in memory and generates a new, random system call number using a secret key stored in kernel space. This requires rewriting the system call dispatcher in the kernel to decrypt the system call numbers at runtime.

For ASLR, it implements library re-mapping and function randomization. Library re-mapping randomizes the library base addresses and reorganizes the internal functions. This makes it more difficult to predict both the absolute and relative addresses. The import and export function tables used by the dynamic linker are also randomized. The function randomization makes the name-lookup of each function unique to each process. Different randomization algorithms are used depending on whether the function is being imported or exported. Due to this, a separate function name resolver needs to be created to tie the imported and exported function names back together at runtime.

Additional protections are also implemented with RandSys. Decoy entries are placed in the function import and export tables. Each decoy points to a guard page which will cause an access violation exception if there is an attempt to read, write, or execute it. RandSys also implements a method for dynamic injection detection. A code page with injected shell code will have two properties that can be detected: it will be writable and it will not be mapped from the executable file. Whenever a system call or library function is invoked, a recursive stack-based inspection algorithm can determine if any of those code pages exist. It hooks into the exception handler and watches for such exceptions. It will attempt to terminate any program that has such an exception.

Entities Protected: All programs running on a machine utilizing this technique.

Deployment: Can be deployed on any generic machine by modifying its OS.

Execution Overhead:

- Increased system call overhead (difficult to estimate but could increase execution overhead by up to 20%).
- Additional overhead introduced by one-time disassembly/analysis of each executable, up to several minutes per executable.

Memory Overhead:

- None

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: The ASLR implemented by RandSys cannot be combined with another ASLR implementation, so one has to be selected. Also, it is desired to combine a solution like RandSys with a ROP protection technique.

Weaknesses: This defense can be circumvented with a ROP attack that can find the location of the randomized libraries (through an independent leakage attack or other violation of memory secrecy or brute force).

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Makes it difficult for an attacker to inject code into a running program and increases the level of effort required to redirect program control to a chosen location.

Availability: This implementation has been prototyped for both Windows and Linux but there is not a publicly available version of it.

Additional Considerations: This technique breaks self-modifying codes. It also requires an additional disassembly step for each application.

Proposed Research: RandSys mainly protects system calls. An extension to RandSys that protects other library calls is an open problem (see [61]). In addition, this type of protection does not prevent ROP attacks. A complete protection against typical code injection and ROP attacks is an open problem.

Funding: National Science Foundation, Microsoft Research

2.2.4 Randomized Instruction Set Emulation

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: ISR

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This method [14, 63] is targeted at stopping external binary code injection into an executing program. The keys used for randomization are stored in the same memory space as the running process so it relies on the assumption that the process memory cannot be read by an attacker.

Description:

Details: Randomized Instruction Set Emulation (RISE) is a software-based ISR technique built on top of the open source Valgrind IA32-to-IA32 binary translator. It scrambles the instruction set at load-time and descrambles them at runtime. It runs in user-space and does not require any modification of the OS or program being run because it is running inside an emulator. It can be run on a per-program basis so it does not interfere with programs like compilers. RISE scrambles all executable portions of a process, including libraries, by XOR-ing each byte of the process' code with a randomization mask. RISE has two methods of randomization. The first method is a tiled method that involves generating a random mask with two or more pages before execution and XOR-ing each byte in the code with a byte in the mask. The mask is read from

/dev/urandom and is stored in a fixed location right before the executable. The second method uses a one-time pad by using a unique mask for each code page. The masks are not generated until the page is first accessed.

In both cases, any code that is injected into the program will be decrypted using the masks and likely result in an invalid execution. For an attacker to circumvent this, he would have to be able to generate a code segment that decrypts correctly into another one with his desired behavior. Ideally, this can only be done if he discovers the encryption keys.

Entities Protected: Any program running inside the RISE emulator.

Deployment: This technique can be deployed on any generic machine by adding an emulator.

Execution Overhead:

- Additional 5% increase in overhead on top of Valgrind overhead.
- Valgrind adds a minimum of 400% overhead per the documentation.

Memory Overhead:

- Each process creates a private copy of all loaded libraries in virtual memory.
- The one-time pad randomization doubles the amount of memory needed for the code.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification required)

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: RISE relies on a emulation layer (Valgrind). If this is to be used for all of the applications, the overhead will be significant.

Weaknesses: This framework does not protect against attacks that target functions or pointers, including ROP attacks. Additionally, an attacker that can violate memory secrecy could read the key directly from memory or recover an encrypted code segment that, along with the unencrypted segment obtained from the original executable, can be used to deduce the key.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique makes it difficult for an attacker to inject viable shellcode into an application running inside the RISE emulator, without having an independent vulnerability that can violate memory secrecy.

Availability: Prototype available under GPL at http://cs.unm.edu/˜immsec.

Additional Considerations: The large overhead introduced by the emulation layer can make RISE impractical for real-world applications. See [57] for a discussion of performance issues.

Proposed Research: Similar to the function pointer encryption technique above, RISE could benefit from the hardware level AES instruction providing an encryption scheme resistant to the known plaintext attack outlined above.

Funding: National Science Foundation, Office of Naval Research, DARPA, Sandia National Laboratories, Hewlett-Packard, Microsoft Research, Intel Corporation

2.2.5 SQLRand

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: ISR

Threat Model:

Attack Technique Mitigated: Code Injection

Details: SQLRand [15] aims to protect against SQL injection attacks in situations where the query depends partially on untrusted input.

Description:

Details: SQLRand is a system for randomizing the SQL query language to prevent SQL injection attacks. The creators note that injection attacks on SQL can be thought of similarly to buffer-overflow-based code injection attacks. Their methods for SQL are based on similar methods in RISE for such attacks. The SQL language is randomized so that any code that was injected will not run (it will not match the new randomized language). A base SQL query (without runtime criteria derived from user input) is sent to a proxy server to be randomized and returned. The randomization is done by appending a chosen integer to the end of every keyword in the SQL language. When the query is executed, it is again sent to the proxy that derandomizes it and passes it on to the database server. Any code that was injected into the query by the user will not match the new randomized language and will cause the query to fail.

Entities Protected: Any database application that uses the SQLRand proxy.

Deployment: Can be deployed on a network as a standalone proxy or on the machine that runs the database software. Requiring use of the proxy to access the database would increase security.

Execution Overhead:

- The randomization is relatively simple and very fast, experimental query response times were increased by 6 milliseconds.

Memory Overhead:

- None

Network Overhead:

- Requires a proxy for all traffic going to the database server.

Hardware Costs:

- Can be run on the same server as the database software, but could also be run as an independent server for increased speed.

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification is required)

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: None

Weaknesses: If the randomized SQL query is ever leaked or accessed by the attacker then he can produce valid injection code. This is very common with web applications that often report the query used upon failure. Developers would have to be very sure that error messages were sanitized and no other paths for query leakage were introduced. However, since most SQL injection attacks start by discovering a query (otherwise the attacker would have no knowledge of the database structure), this seems like a very large weakness.

Types of Weaknesses:

☐ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Increases the level of effort for SQL injection attacks by making it more difficult for attackers to generate valid injection code. It requires them to either brute force the random key or find a way to leak an existing randomized query.

Availability: No code is publicly available.

Additional Considerations: This approach requires every use of a SQL query to be rewritten (in the source code) with this randomization in mind. In particular, the developer must identify the parts of the query that will always remain the same and the parts that are based on user input. Since the vast majority of SQL injection attacks occur because the developer did not take the time to do this in the first place (if he did there is already a method for sanitizing inputs using the prepare command), this seems like a wasted effort. Moreover, the scope of the protection is also very limited.

Proposed Research: There are existing, effective techniques to stop SQL injection attacks. No research is proposed.

Funding: Unknown

2.2.6 Against Code Injection with System Call Randomization

Last Updated: 6/29/2012

Defense Category: Dynamic Runtime Environment

Defense Subcategory: ISR

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This technique [16] protects against injection of code into an application with a buffer overflow vulnerability. This technique is only effective against injected code that requires the use of system calls.

Description:

Details: First, the compiler is modified so that each system call number is changed from x to $f(r, x)$ where f is a random permutation that takes r as an input seed. In practice, they use XOR as f. This means that every system call number is replaced by a randomly chosen pseudonym. Any code that is injected will not know this mapping and thus cannot produce shellcode that invokes the correct system call. The kernel system call dispatch is changed so that it knows f and r and can derandomize the input number to the correct system call number.

Entities Protected: Any programs recompiled using the modified compiler on a system that includes the kernel derandomizer.

Deployment: Can be deployed on any generic machine by modifying the OS.

Execution Overhead:

- The kernel must derandomize system call numbers, but system call dispatch already takes a significant amount of time and one additional XOR does not have significant impact.

Memory Overhead:

- None

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

Kill Chain Phases:

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☒ Attack Launch
- ☐ Persistence

Interdependencies: Could be combined with ASLR to increase protection.

Weaknesses: The system call table is not very large so the amount of randomness introduced is small (can be as low as 8 bits). Additionally, if a randomized binary is leaked then an attacker can compare that to a regular binary and discover the key, gaining the ability to forge system call numbers. Also does not protect against return-oriented attacks because the system calls in libc will already be correctly randomized.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Increases the effort required to inject the desired code successfully.

Availability: No code publicly available, but the concept is relatively simple and would not require many code changes.

Additional Considerations: None

Proposed Research: As with other techniques that use XOR as an encryption function, this could possibly benefit from hardware AES. This may be a better research opportunity because system calls happen relatively infrequently (compared to pointer dereferences) and already require a shift to kernel space. This means that any performance degradation will be well hidden and the problem of storing keys is dealt with because they can be securely stored in kernel space. However, this solution is a partial solution to a bigger problem. A proper memory protection against regular code injection and ROP is required.

Funding: National Natural Science Foundation (China), Beijing Science Foundation, Nation 868 High-tech Program of China, MOE Key Laboratory of Data Engineering and Knowledge Engineering

This page intentionally left blank.

3. DYNAMIC SOFTWARE

3.1 SOFTWARE DIVERSITY USING DISTRIBUTED COLORING ALGORITHMS

Last Updated: 6/29/2012

Defense Category: Dynamic Software

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This technique [17] reduces the number of machines an attacker can successfully compromise in a network using code injection attacks. It does not prevent individual machines from being compromised.

Description:

Details: This meta-technique involves taking existing code diversity techniques and applying them across an entire network. The authors attempt to answer the following question: assuming that an adversary must specially craft an attack for each version of a diverse executable, and we have access to *k* versions of an executable, how can we place these versions on a network so as to minimize the number of compromised machines (conversely, maximize the effort of the attacker)? Since we are trying to minimize the number of connected machines running the same version, this is the same as asking for an optimal *k*-coloring of the graph representing our network. Unfortunately, finding the minimum number of colors needed for a perfect coloring of a graph (such that no connected nodes are the same color) is NP-hard, as is the problem of finding an optimal coloring using *k* colors. Instead, they propose a distributed heuristic approximation algorithm that results in at most *n/k* links between two nodes of the same color, where *n* is the number of nodes. If $k \geq n$, then each node can have its own color (version of the software) and the attacker will require a new custom attack for each node. If it is lower, then an attacker will only be able to infect a new node at each step with probability approximately equal to 1/k. This gives us good utility out of the diverse executables that we do have.

Entities Protected: The overall network is protected from easy compromise by an attacker.

Deployment: The approximation algorithm used for assigning versions is distributed meaning that it must be run on every computer in the network. It could also be deployed from a centralized server that is distributing software to the network.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification is required)

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This technique relies on already having diversified versions of the applications available. Other diversification techniques must be available for this technique to be useful.

Weaknesses: The proposed idea is more a planning tool than a stand-alone technique. Also even assuming that diversity can stop large-scale attacks, this method does not stop attacks against one machine.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: If the underlying diversity used is sound, then this technique makes it harder for an attacker to compromise an entire network using only one attack. Depending on the number of software versions available, he could be limited to a small portion of the network.

Availability: None, results only theoretical.

Additional Considerations: This is more a planning method that a stand-alone technique. The results are highly theoretical.

Proposed Research: The actual impact of diversity on successful attacks must be studied and analyzed.

Funding: National Science Foundation, Koerner Family Fellowship

3.2 SECURITY AGILITY FOR DYNAMIC EXECUTION ENVIRONMENTS

Last Updated: 6/29/2012

Defense Category: Dynamic Software

Threat Model:

Attack Technique Mitigated: Exploitation of Trust

Details: This technique [18] aims to mitigate system and network intrusions at a high level by dynamically modifying security policies.

Description:

Details: The authors describe and implement a software toolkit that allows applications to be developed around the idea of dynamically changing security policies. The main problem with moving from static security policies to dynamic policies is that unmodified applications will not be able to adjust to policy changes that leave them without access to crucial resources. The authors introduce a framework for designing applications with multiple behaviors that can transition from one to another depending on which resources (both on the same machine and on the network) are available under the current security policy. This allows security policies to change on the fly, in response to an actual or attempted intrusion, while maximizing the utility of the machines and applications on the network at all times. An agile policy controller that can set and modify the security policies over the whole network dictates the security policies on each machine.

Entities Protected: This technique protects the network from potential intrusions and provides a way of mitigating successful intrusions.

Deployment: This technique requires deployment on all machines in a network as well as at least one additional policy controller.

Execution Overhead:

- Varies depending on the application; backup behaviors could be less efficient in order to get around reduced resources of some security policies.

Memory Overhead:

- Varies depending on the application; backup behaviors could be less efficient in order to get around reduced resources of some security policies.

Network Overhead:

- Varies depending on the application; backup behaviors could be less efficient in order to get around reduced resources of some security policies.

Hardware Costs:

- Requires at least one additional policy controller machine.

Modification Costs:

☐ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☐ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This technique relies crucially on a detection capability. This can be very challenging for polymorphic type attacks [94, 95]. If the attacks are not detected, they cannot adjust the policy.

Weaknesses: The policy manager becomes a new point of weakness since it can dynamically change the security policies of all the other machines on a network. The authors provide a mechanism for distributing the duty amongst several machines so that no single point of trust exists.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: Makes it more difficult for an attacker to advance an intrusion due to the network security policies reacting dynamically to his attack.

Availability: Research was done as part of a Defense Advanced Research Projects Agency (DARPA) project, so we assume the code is available.

Additional Considerations: This work lacks many specifics. For example, how the policy is adjusted or what impact policy adjustment has on the system. See [71] for more on dynamic policy.

Proposed Research: The actual impact of agility and policy adjustment on the security posture of a system must be studied. Also, reliance on a perfect detection capability must be relaxed in such a system.

Funding: DARPA

3.3 PROACTIVE OBFUSCATION

Last Updated: 6/29/2012

Defense Category: Dynamic Software

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [19] aims to mitigate buffer overflows and other injection attacks on network visible services.

Description:

Details: The authors use a similar technique to DieHard but in a more generalized setting. Since control injection attacks have to be individually tailored to specific executables, this technique creates multiple copies of each service executable, randomized differently. The randomization used can be any of the other executable randomization techniques we have described such as ISR, ALSR, or system call randomization. Whenever a request is issued to the service, it is multiplexed to each of the replicas and the responses are tallied like a vote. If a majority of the replicas agree, then the response is sent out. The idea is that any attack should only work on one of the replicas and the others will remain uncorrupted, so a majority vote will result in a correct response. However, it is more likely that one will be compromised and the others will crash (due to different addresses, system calls, etc.). This means that if the system returns a response, it will be correct with a high degree of certainty but it may not answer if a majority of the replicas have crashed. In order to prevent an attacker from gaining some progressive knowledge and eventually letting him compromise all the replicas at once, the system proactively reboots replicas with new randomization. There is a controller that dispatches the requests and tallies votes, as well as controls when replicas will be rebooted (a configurable time limit).

Entities Protected: Protects servers.

Deployment: Can be deployed on any server with important trusted services.

Execution Overhead:

- Experimental execution overhead of 20% (differs depending on application, this estimate is very optimistic) and latency overhead of 40%.

Memory Overhead:

- Extra memory must be used to store the multiple running replicas so an *M* times memory overhead where *M* is the number of replicas.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

- ☐ Data
- ☐ Source Code
- ☒ Compiler/Linker
- ☒ Operating System
- ☐ Hardware
- ☐ Infrastructure

Expertise Required to Implement:

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☐ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

- ☐ Seamless
- ☒ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

Kill Chain Phases:

- ☐ Reconnaissance
- ☐ Access
- ☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This method does not propose a new randomization technique and relies on existing diversification techniques.

Weaknesses: The controller that dispatches and maintains replicas is now a new target for attack, since it is a single point of failure. Additionally, a single compromised replica can destroy it if it is not also replicated. Also, this technique does not protect against information leakage (exfiltration) that happens on one replica.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: Makes it difficult for attackers to cause services to return incorrect results.

Availability: No publicly available code.

Additional Considerations: The technique on its own does not provide protection. It relies on existing randomization techniques and voting.

Proposed Research: This technique ensures correct responses by voting amongst the replicas, but it does not ensure that individual replicas cannot cause damage locally. If multiple replicas were running on the same machine, the OS interface (system calls) could be considered as the other side of a container holding these replicas. Every time a single replica executes a system call, if the other replicas are uncompromised they will also issue the same system call. If one of the replicas is compromised, it must deviate from proper behavior by calling a different series of system calls that can be detected as aberrant. If it does not deviate, then it cannot do anything useful. Therefore, the OS could only execute system calls if a majority of the replicas request the same system call, ignoring all others.

Funding: Air Force Office of Scientific Research, National Science Foundation, Microsoft Corporation

3.4 PROGRAM DIFFERENTIATION

Last Updated: 6/29/2012

Defense Category: Dynamic Software

Threat Model:

Attack Techniques Mitigated: Control Injection and Code Injection

Details: This technique [20] mitigates buffer overflow attacks on remote services.

Description:

Details: The authors aim to design a secure mobile phone platform that is not vulnerable to remote attack through buffer overflow exploits. They note that buffer overflow attacks can be defended against using several different orthogonal techniques to increase effectiveness. One of these techniques, system call randomization, is old, and two more are unique to this report.

The authors propose that, since mobile platforms are rapidly evolving, it may be useful to consider hardware changes that could defend against buffer overflow attack. Toward this, the first defense they propose is modifying the return instruction. The vulnerability in the return instruction is that it returns to an address specified on the stack, which can be targeted by an attacker. Instead, the new return address will only take an index into a table that contains the actual return addresses. This table will be readable only by the return instruction and writable only by the call instruction, so it will not be vulnerable to inspection or tampering. At the start of a function call, the call instruction will insert the return address into this table with a random unused index. It then puts this index on the stack. The return instruction loads the actual address from the table based on the index on the stack and jumps to the specified location. The address table is protected so that it can only be read by the return instruction and written to by the call instruction.

The second technique the authors propose is to use instruction packing to differentiate at the instruction set level. The way instruction packing works is it compresses frequently used instructions together into one instruction with an Instruction Register File (IRF). This IRF stores the instructions in an indexed table and when the program wishes to use a sequence of these instructions it can instead call a 5-argument pack instruction with the indices of the instructions it wishes to use. For instance, if an often used sequence of instructions is stored in the table with indices 1–5, the program would invoke all five instructions at once with a single instruction *pack5 1 2 3 4 5*. If the indices of the IRF are randomized then this creates a unique instruction set for each executable.

Entities Protected: This scheme is targeted at mobile platforms but could be used anywhere the custom hardware was available.

Deployment: Deployed at the local machine level by modifying hardware.

Execution Overhead:

- Unknown execution overhead due to additional table lookups.

Memory Overhead:

- None

Network Overhead:

- None

Hardware Costs:

- Requires special hardware with the modified instruction set described.

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☒ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This method should be combined with a ROP defense.

Weaknesses: The method is vulnerable to ROP without returns since the jump instruction is not protected.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: Makes it very difficult for an attacker to inject code (since he cannot guess the correct indices into the IRF) and impossible for an attacker to return to arbitrary locations in the code. ROP is still possible though.

Availability: The hardware specified does not actually exist yet.

Additional Considerations: The technique is effective against traditional code injection, but the hardware modification proposed makes it impractical for existing systems.

Proposed Research: A complete code injection and ROP protection method is an open problem.

Funding: National Science Foundation

3.5 REVERSE STACK EXECUTION IN A MULTIVARIANT EXECUTION ENVIRONMENT

Last Updated: 8/6/2012

Defense Category: Dynamic Software

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This technique [21, 50, 52, 53, 54] detects buffer overflows on the stack and prevents exploitation of them through stack smashing.

Description:

Details: The authors propose a very simple form of multivariant execution with two replicas where one replica runs with the stack growing upwards and the other runs with the stack growing down. Normally any single architecture only supports the stack growing in one direction, but the authors introduce a compiler transformation that can create a program with an opposite direction stack. Any buffer overflow attack that works on one would necessarily not work on the other because the overflow would be writing over different parts of the stack. Therefore, a divergence in behavior would signify that such an attack has occurred and the OS could detect that and terminate the program.

Entities Protected: Any generic machine with this technique deployed in the compiler.

Deployment: Deployed on any machine by modifying the compiler and OS.

Execution Overhead:

- 100% execution overhead to run a replica.
- Experimental results show only a 3% overhead in the replica.

Memory Overhead:

- Up to 20% increased executable size.
- 100% memory overhead for an additional replica.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This method should be combined with ASLR and ROP protection techniques for better results.

Weaknesses: A monitor is required to dispatch inputs to both replicas and to detect when their execution diverges. This monitor is itself vulnerable to attack as it has the same weaknesses as any other program. Additionally, there are some special cases where a buffer overflow can work on a stack in both directions equally. Specifically, if a buffer overflow occurs and there is no system call between it and the return function.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: Attackers must find a weakness in the monitor or a more specific type of buffer overflow.

Availability: No code publicly available.

Additional Considerations: It requires source code of any application to be protected. It also requires an additional replica to be run (100% execution overhead). Similar, but more limited multi-variant techniques have been proposed [51].

Proposed Research: An improved technique can use a similar method but without relying on replicated execution.

Funding: Unknown

4. DYNAMIC NETWORKS

4.1 DYNAMIC NETWORK ADDRESS TRANSLATION

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Techniques Mitigated: Scanning, Resource, Spoofing, and Data Leakage

Details: This technique [22, 23] assumes the hosts and entities employing this technique are safe. It can help mitigate scanning attacks by obfuscating various parts of network packet headers but not the payload of the packets. Depending on the placement of the obfuscator, it could be used to combat some resource attacks like denial of service attacks. If the attacker is sending a flood of packets, the protected packet fields would be unencrypted and produce random values which would likely result in them not hitting the intended service. This same property would also increase the likelihood of detecting anomalies. This technique would also increase the difficulty of performing some spoofing attacks. It would be more difficult for an attacker to capture some traffic and replay it back to the service because of the changing obfuscation keys and uncertainty about how the network is currently mapped.

Description:

Details: Dynamic Network Address Translation (DYNAT) is a protocol obfuscation technique. The idea is to randomize parts of a network packet header. This randomization can make it more difficult to determine what is happening on a network, who is communicating with whom, what services are being used, and where the important systems are located depending on how the technique is deployed. Some parts that can be scrambled include the Media Access Control (MAC) source and destination address, IP source and destination address, IP type of service (TOS) field, Transmission Control Protocol (TCP) source and destination port, TCP sequence numbers, TCP window size, and the User Datagram Protocol (UDP) source and destination port. Ideally, the randomization is done with a strong cryptographic hashing scheme or encryption. The key can be changed on a clock-based scheme or via properties in the network such as packets sent. The key used to scramble can be generated via static means on each host, it can be split to be partially static and partially locally or externally dynamic, or it can be fully locally or externally dynamic.

Entities Protected: This technique aims to protect the network traffic as it is traveling between systems.

Deployment: This technique can have a number of different deployment scenarios depending on the level of protection needed. It can be deployed to workstations, servers, routers, and gateways. This could be used to protect switched local area network (LAN) segments, contention-based LAN segments, LAN-to-LAN connections (local router connections), Gateway-to-Gateway connections (networks separated by the internet or long range connection), or a combination of LAN segments and gateway connections.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- Depending on the deployment and fields obfuscated, the network overhead can be significant. For instance, if using this on a switched network and obfuscating the MAC address, this could cause the switches to fill up their memory and cause a lot more Address Resolution Protocol (ARP) traffic to determine which switch port to route packets through next.

Hardware Costs:

- Additional hardware may be required to handle the routing overhead.

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☒ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: To make this technique more effective, it should be used in combination with a packet payload encryption mechanism. Possibilities might include Secure Sockets Layer (SSL) or the IP Security (IPSec) protocol. Another mechanism needed is a reasonably strong encryption mechanism for the protocol obfuscation. A mechanism to generate new keys securely across all the participating systems is also necessary.

Weaknesses: The use of other networking protocols can reduce the effectiveness of this technique. Additional information is added to the packet headers with protocols like Multi-Protocol Label Switching (MPLS) or using static virtual local area networks (VLANs). This additional information cannot be obfuscated and would leak additional information about what is going on inside the network. This technique does not do anything to change packet sizes, vary packet timing, or use dummy packets so it is susceptible to traffic analysis. More importantly, this technique only limits reachability. For services that can be reached from outside, this technique offers no protection.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique increases the workload for an attacker but does not necessarily stop them from collecting the information they need. Traffic analysis could still be used to profile types of traffic or the payload of the packets could be analyzed to collect information about the network.

Availability: This was prototyped by the original authors but is not publicly released.

Additional Considerations:

- This technique can severely limit a server's functionality because it cannot be reached from outside.
- Depending on the placement of these obfuscators, it could have adverse effects on other network equipment. For example, placing them behind routers or gateways may inhibit that device's ability to do traffic filtering.
- Depending on the fields obfuscated and the placement of the obfuscators, it could have adverse effects on other network equipment. For example, if MAC address obfuscation is being used, it could break port locking on switches if the MAC address is changing constantly due to obfuscation key rotation.
- Depending on the fields obfuscated, it could have adverse effects on other network protocols. For example, if MAC address obfuscation is being used, it could break dynamic VLANs.

Proposed Research: This technique could be expanded to harden it against traffic analysis techniques. The obfuscators could be modified to include additional scrambling. This could include varying the timing of packets are sent from the system, inserting extra padding into the packets to vary packet size, and sending out dummy packets. Payload encryption is not currently a part of this technique and it increases the effectiveness of the technique by not allowing the attacker to analyze the content of the packets. More research would be needed to determine if there are more cases of special protocols leaking information making this technique less effective.

Funding: Sandia

4.2 REVERE

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Techniques Mitigated: Resource, Spoofing, and Data Leakage

Details: This technique [24] can help protect against a couple of classes of attacks to some degree. It helps protect against resource attacks like denial of service or manipulating content on the network. The effects of denial of service attacks are mitigated by the distributed and well-connected nature of the overlay network. An attacker would need to be able to flood potentially many thousands of machines simultaneously. This technique helps protect against content manipulation by using digital signatures on the content that it is distributing. This allows every node in the network to verify the content assuming the signature has not been compromised. This technique also helps protect against some spoofing attacks like man-in-the-middle, traffic replay, and impersonation attacks. This would help mitigate man-in-the-middle attacks by using strong authentication and trust relationships between each node in the network. Content replay attacks are mitigated by dropping duplicate content at each node. Impersonation attacks are also mitigated by the use of public key cryptography and digital signatures.

Description:

Details: Revere is a technique that involves creating an open overlay. An overlay network is an example of a dynamic network in that it can change paths, reconfigure, and respond to links or nodes going down dynamically. The network consists of a central distribution center that is the root of the network and nodes, or clients, receiving the content from the distribution center. Each node in the network can be a parent or a child. A parent can have multiple parents and multiple children. When a new node wants to join the network, it determines the fastest parent that it can attach to and performs a handshake with that parent to see if it will accept the new node. Once a node has found a parent, it then seeks out other additional parents to increase its resiliency.

Security is accomplished in this overlay by the distribution center digitally signing the content it is pushing out. Each node in the network can verify the signature of the content before using it and passing it on to its children. If the authenticity of an item is in question, it can be pushed back up to the node's parents and eventually the distribution center to be verified. Security can also exist between the parent and child nodes. Each node can support some set of authentication methods and the child can negotiate a method with the parent. Security appliances or authorities can also be employed for this task such as a Certificate Authority. Each node can have its own set of rules to determine if it should trust a parent or a child when they are negotiating.

Reliability is accomplished by the many-to-many relationships between the nodes. This provides many paths for content to be delivered and duplicate items are dropped at a node. Each node employs a heartbeat type message between its parents and children to determine if they are still online. If a child does not receive a heartbeat from a parent in a certain amount of time, it will assume that parent is gone and not use it anymore. Each parent is also capable of sending a message to tell a child that it is no longer usable.

Fast delivery is accomplished by each node maintaining the fastest path back to the distribution center. Each child has a Parent Path Vector (PPV) that is the fastest path back to the distribution center, which includes that parent. It also maintains a Node Path Vector (NPV) that is the fastest of the PPVs. If a parent that is part of the NPV goes down then the next fastest of the PPVs is chosen to be the new NPV. The speed of a link can be calculated at the child by analyzing the timestamps of the periodic heartbeat messages. The mesh-like distribution of nodes also helps push content out quickly.

The technique was prototyped as a Java client and tested up to 3000 nodes. Their testing showed that an update could reach all nodes in less than one second on average. They projected that an update could reach every node on a network of one hundred million nodes in less than four seconds.

Entities Protected: This technique protects the integrity and availability of content delivered over a network.

Deployment: This technique would be deployed as a client on a system that wishes to participate in the overlay.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- The control messages passed between nodes does cause extra traffic on the network. Reconfiguration and routing can impose unknown network overheads.

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification is required)

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This technique relies on good authentication mechanisms between nodes, good rules to determine if a node is trustworthy, and the security of the distribution centers.

Weaknesses: The security of the updates relies on the security of the distribution center. The authors mention that there are backup private keys available to use if one is compromised but, if an attacker is able to compromise one, it is not unreasonable to conclude he could compromise the backups as well. This would allow the attacker to masquerade as the distribution center and push out fake updates, pollute the update repositories, or do other tampering of the content. The node trust mechanism suffers from a similar problem. If keys are being used to sign messages to determine the authenticity and trustworthiness of a node, an attacker could have compromised a previously trusted host and use their identity. Also more importantly, the technique is focused on protecting reachability, if the machine can be reached from outside the overlay network, this technique does not provide any protection.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: The amount of impact this has on an attacker depends on the attacker's goals. If the attacker intends to poison the network with malicious or corrupted updates, then the impact is correlated to the difficultly of compromising the distribution center and the private keys. If the attacker were attempting to bring down the network, the increase in difficultly would be correlated to the size of the network. The larger and more connected the network is, the more difficult it would be for an attacker to disrupt it as a whole. However, this technique does not provide protection for individual hosts.

Availability: This technique was prototyped by the authors but is not publicly available.

Additional Considerations: Some aspects of the paper are left very vague. Having a large trusted network or authentication between all nodes in the network is a good idea, but if the network is spread across the world, how is setup for a new node wanting to join the network done? It discusses setting trust rules for a node but it is not clear what such a rule would entail or how a system could determine if another node is truly trustworthy simply by a handshake request.

Proposed Research: A dynamic network solution combined with other host protection techniques must be explored.

Funding: Unknown

4.3 RANDOMIZED INTRUSION-TOLERANT ASYNCHRONOUS SERVICES

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Technique Mitigated: Resource, Exploitation of Privilege/Trust, Scanning

Details: This technique [25] is meant to impede an attacker from manipulating messages on the network or taking a service offline. The proposed protocols allow various processes to reach agreement on a message while accounting for a certain threshold of bad processes. Running multiple instances of processes also creates more redundancy in the service.

Description:

Details: Randomized Intrusion-Tolerant Asynchronous Services (RITAS) is a technique that builds a set of fault-tolerant consensus-based protocols on top of TCP and the IPSec protocol. TCP provides a reliable channel and IPSec provides integrity to the data being transmitted. This technique is to be used between a set of n processes. A process is considered corrupt if it does not follow its protocol until termination. This technique can handle at most $f = \left\lfloor \frac{n-1}{3} \right\rfloor$ corrupt processes. There are no assumptions about bounds on processing times or communication delays. The processes are assumed to be fully connected and each pair of processes shares a secret key.

The first protocol is reliable broadcast. This protocol ensures that all correct processes deliver the same message and, if the sender is correct, the message is delivered. The next protocol is echo broadcast. It is a more efficient and less powerful version of the first protocol. It does not guarantee all processes will deliver a message if the sender is corrupt. The first consensus protocol is binary consensus. It builds upon reliable broadcast and allows processes to agree on a binary value (either one or zero). It is the only protocol of this technique that includes randomization if a consensus cannot be made. The next consensus protocol is multi-valued consensus. This allows the processes to agree on arbitrary length values and builds on top of reliable broadcast, echo broadcast, and binary consensus. The next consensus protocol is vector consensus. It allows the processes to agree on a subset of proposed values. It builds on the reliable broadcast and multi-valued consensus protocols. This ensures that each process decides on the list of values of size equal to the number of processes. Each element of the list corresponds to a process (element one of the list is the value of process one and so on). This ensures that each element of the list is either the value proposed by that process or the default value and at least $f + 1$ elements were proposed by correct processes. The final protocol is the atomic broadcast protocol. This protocol builds on reliable broadcast and multi-valued consensus. This protocol ensures that each message is delivered reliably and in the same order to all processes.

Using randomization, this technique implements a dynamic network that is capable of guaranteed delivery given limited number of malicious nodes.

Entities Protected: This technique protects the information returned from a service by ensuring a majority of the services agree on the results.

Deployment: This technique would be integrated into the code of programs that wanted to use these new protocols.

Execution Overhead:

- Additional resources required to run many of the same services.
- Additional time added while the protocols are reaching agreement.

Memory Overhead:

- Additional resources required to run many of the same services.

Network Overhead:

- IPSec adds an additional 24 bytes to each packet header.
- Additional network traffic by all the broadcasting and exchanging of messages while the protocols are reaching agreement.
- IPSec adds an average 30% latency for each protocol.
- Can have an impact on network throughput for large volumes of traffic.

Hardware Costs:

- None

Modification Costs:

☐ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: There are limited applications that can benefit from a protocol like RITAS. Additional protection techniques are certainly necessary.

Weaknesses: One large weakness of this technique is that is can only tolerate $f = \left\lfloor \frac{n-1}{3} \right\rfloor$ compromised processes. More importantly, RITAS does not provide any protection against one-node compromises. Attacks like data leakage (exfiltration) are still a concern.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique is only useful against integrity attacks. If an attacker were trying to manipulate the output of programs or the data being passed around on the network, this would increase his workload because the attacker would need to compromise a certain percentage of processes as opposed to just one. This also adds additional impact to the attacker if he were trying to take down the process or service. Running multiple copies provides overall greater resiliency if one or more were to fail.

Availability: This technique was prototyped by the author but the code was not publicly released.

Additional Considerations: This method is very limited in scope in that it deals with a particular problem (message passing) with a protocol. Much of this work is very theoretical.

- May not work well for applications that require very low latency or streaming.
- Needs to work with applications where multiple instances would produce the same output.
- Running potentially numerous instances of a process will likely increase the maintenance workload and overhead.

Proposed Research: This technique abstracts out what the processes are actually doing or how they are setup. Adding randomization or diversity techniques to the individual processes or machines they reside on would further increase the workload of the attacker assuming that such diversity did not result in the processes producing different outputs. If all processes were running on similar systems, if an attacker was able to compromise one, he may also be able to compromise many with similar methods degrading the effectiveness of this technique.

Funding: European Network of Excellence, FCT (Portugal)

4.4 NETWORK ADDRESS SPACE RANDOMIZATION

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Technique Mitigated: Resource and Scanning

Details: This technique [26] was designed to mitigate and slow the effects of an IP address hitlist-based worm. It does not actually stop any specific attacks. It can be used on some level to reduce the effectiveness of scanning attacks. The information collected from these attacks would change as the IP addresses of the systems changed.

Description:

Details: Network Address Space Randomization (NASR) is a technique that involves changing the IP address of systems more frequently. The authors modified a Dynamic Host Configuration Protocol (DHCP) server to have short IP address leases and to force an IP address change when a lease expires. The side effect of changing these IP addresses constantly is that persistent or active connections would be dropped during the address change. The authors developed sensors to attempt to profile the services on a system and the connections on a system. If a system has many connections that would be dropped, the changing of the address is delayed. There is a hard limit where a system will be forced to change its IP address as well if it has not changed for a long time. There are some types of systems that have constant persistent connections that would be excluded from this technique. There are also some systems that require a static IP address that would be excluded as well. Domain Name System (DNS) servers can be used for outside access to servers and services to mitigate the impact a constantly changing IP address would have on end users.

Entities Protected: This technique helps mask the identify of systems and servers from information collection and targeted attacks.

Deployment: This technique would generally be implemented in segments of a LAN.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- Dropped connections due to IP address changes during interactions.

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This technique only slows down certain types of attacks but relies on other detection mechanisms to detect these attacks.

Weaknesses: This technique does not protect systems that rely on static IP addresses or systems that use DNS. If a system is using DNS, the attacker can just point to that address and does not have to worry about the actual IP address. The effectiveness of this technique is also reduced if there is not a large enough pool of IP addresses available.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: This may impose some overhead on the attacker to maintain a mapping of systems but it, by itself, does not stop an attacker from launching any attacks against a system or server.

Availability: This technique was prototyped by the authors but code was not publicly released.

Additional Considerations: This technique does not provide any protection against targeted attacks or attacks that can reach the machine using higher level protocols. It also does not protect against client-side attacks (e.g., browsing to a malicious website). The technique is very limited in scope and can break many functionalities.

Proposed Research: This technique could be extended to have a larger pool of addresses to use for randomization. Another idea would be to extend it to randomize network properties such as port numbers. An external abstraction layer or proxy could be used to translate addresses coming into this network such as a Network Address Translation (NAT) device. This would make the individual internal systems transparent to the outside world. Combining this technique with other network technologies that manage connections between systems could reduce the amount of dropped connections due to an address change.

Funding: European Commission\Information Society Technologies, Greek Secretariat for Research and Technology

4.5 MUTABLE NETWORK

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Techniques Mitigated: Resource and Scanning

Details: In this technique [27], the shifting IP addresses would make it more difficult for an attacker launching denial of service type attacks against individual systems in the network. The shifting IP addresses, port numbers, and packet routes would also make it more difficult for an attacker running scans on the network trying to identify what systems are there as well as the services running on those systems.

Description:

Details: A Mutable Network (MUTE) is a technique that involves changing IP addresses, port numbers, and routes to destinations inside of a network. This technique is proposed to be implemented as a sort of virtual overlay to the existing network so the original IP address and information on the systems never changes. All traffic is routed independently over this virtual overlay. Synchronization of IP address information across the network would be done across encrypted channels. There would also be mechanisms in place to apply transformations on the network traffic to confuse the tools attackers are using to identify the services and hosts. The packets can be changed based on rules distributed amongst routing entities. It can change the source and destination IP address as well as source and destination ports. There is a sense of possible network configurations so packets can be rerouted to get to their destination via a different path. There would also be policies in place to ensure any global network requirements are satisfied.

Entities Protected: This technique helps mask the identities of systems inside of a network. By changing the information associated with systems, information collected by attackers would be constantly shifting.

Deployment: This would be deployed on all devices capable of routing network traffic and wish to participate in this technique.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- There may be unknown, but significant overload of network infrastructure including routers and switches.
- The extra routing overhead may break the network infrastructure.

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

- ☐ Seamless
- ☐ Simple Configuration
- ☒ Complex Configuration (System Admin)
- ☐ Expert Operator

Kill Chain Phases:

- ☒ Reconnaissance
- ☒ Access
- ☐ Exploit Development
- ☐ Attack Launch
- ☐ Persistence

Interdependencies: This technique can be combined with other network-based detection and monitoring systems.

Weaknesses: One potential weakness of this technique would be if an attacker could still attack the original IP address of the machine since it does not change. Another possible weakness is if the IP address information does not change fast enough. An attacker could do enough reconnaissance to figure out what they need then launch their attack before the change happens. In addition, if any systems are using a DNS address, this will be updated with the IP addresses and an attacker could target a machine via that address. More importantly, this technique only protects reachability. It does not provide any protection against client-side attacks (e.g., browsing to a malicious website).

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: This technique could have varying levels of impact on an attacker depending on what the attacker is trying to accomplish. If an attacker is trying to disrupt the network, it could make flooding attacks more difficult if they did not have access to DNS addresses. Since this technique is not supposed to disrupt active connections, if an attacker can collect the information they need before a switch and establish a connection to the system, they may not be as impacted as much by this technique.

Availability: This is a research idea and was not implemented.

Additional Considerations: A technique like this could have impact on applications or services that require constant connections or could disrupt current connections. More importantly, the protection offered is very limited. It does not protect against client-side attacks. The technique can also have severe scalability issues.

Proposed Research: Since this is a proposed idea, many aspects are still undefined. It is not clear how the technique could be put in place such that it would not affect active connections or running services. It is also not clear how to handle adding or removing systems from this network or how far it would scale. It must also come up with a way to be fast enough to impact attackers while not overloading the systems or network with the changes. Finally, how this needs to be implemented into a system would need to be investigated as well. It is not clear how the underlying actual network is protected if at all. If an attacker is still able to get in through the original network that does not change, then it defeats the purpose of this technique.

Funding: Unknown

4.6 DYNAMIC BACKBONE

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Technique Mitigated: Resource

Details: This technique [28] is designed to mitigate a specific type of resource attack known as denial of service. It does this by dynamically rerouting network traffic away from virtual overlay networks that are being flooded.

Description:

Details: Dynamic Backbone (DynaBone) is a technique that involves creating multiple inner virtual overlay networks inside of a larger outer virtual overlay network. Each of the inner networks can be using a different networking and routing protocol or hosting a different service to increase diversity amongst them. Each host in the outer overlay network is not aware of the inner networks giving the appearance of only one network. The entry points to these internal overlays have a collection of sensors that monitor performance and possible attack traffic. Based on the conditions of the networks, it decides which internal network to use. If an internal overlay is detected to be under attack or is suffering performance issues, traffic can be routed through different overlays (dynamic network aspect of DynaBone). This technique is built on top of X-Bone that is a dynamic network overlay technique that allows multiple simultaneous virtual overlays to coexist. It allows network topologies to be dynamically created and used by applications. Hosts and networking devices can participate in multiple overlays. This can also be setup so various physical paths in the network are unique to different overlays.

Entities Protected: This technique aims to protect the availability of services on a network. Traffic can be dynamically rerouted or routed through multiple paths simultaneously.

Deployment: This would be deployed on all entities participating in the virtual network.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- Depending on the various networking and routing protocols that are being deployed with this technique, they can add additional latency and reduce bandwidth. These can include encryption and authentication protocols/algorithms.

- The impact of additional routing and load on the network infrastructure is unknown.

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: A good detection mechanism to detect when an overlay is under attack. This technique assumes that attacks can be detected.

Weaknesses: The inner overlays may not be sufficiently disjoint and it could be the case that the loss of certain hosts/networking devices/routes can severely affect the overall network. If the service is not distributed, it is also possible for an attacker to take the service out by flooding the service provider. Also this technique does not provide any protection against targeted attacks or data leakage (exfiltration) attacks. This technique does not provide any protection against client-side attacks.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: This technique has varying levels of impact on an attacker depending on the goals of the attacker. If an attacker is trying to take down a service by flooding hosts or network infrastructure, this technique could make it more difficult for him. If an attacker were attempting to take out a service via other means such as attacking the service directly, this technique would be less effective.

Availability: This technique was prototyped by the authors but the code was not publicly released.

Additional Considerations: This technique is limited in the protection it provides. It does not provide any protection after a host is reached. More importantly, it does not protect against client-side attacks. In addition, this technique can severely impact functionality by limiting communication.

Proposed Research: One idea for this technique is go combine it with techniques that also increase the resiliency of the end service as well. This could include techniques that run multiple instances of a service. This would increase the overall availability of the service by making it more difficult for the attack to disrupt the network and the end service.

Funding: DARPA, Air Force Research Laboratory

4.7 ACTIVE REPOSITIONING IN CYBERSPACE FOR SYNCHRONIZED EVASION

Last Updated: 6/29/2012

Defense Category: Dynamic Networks

Threat Model:

Attack Techniques Mitigated: Scanning and Resource

Details: This technique [29] helps mitigate scanning related attacks by continually changing IP addresses. Hopping makes the life of the collected information limited. This technique would also help mitigate some resource attacks related to denial of service (DoS) attacks. It is presumed that the gateways do not have a global DNS address so an attacker would need to target a large set of IP addresses simultaneously or constantly change the target for a DoS attack to reach the target.

Description:

Details: Active Repositioning in Cyberspace for Synchronized Evasion (ARCSYNE) is an IP address hopping technique implemented at VPN gateways. The functionality is implemented into the kernel of the gateway OS. Each gateway participating in the hopping shares a secret and a clocking mechanism. At each clock tick, the gateways compute a new IP address based on the secret and the clock. Each gateway also computes what the other gateway IP addresses will become. The IP hopping does not disrupt connections between gateways including streaming services. In order to account for packets that are delivered shortly after an IP address change, the gateways can still accept those packets up to a grace period. This grace period should be approximately equal to the time it takes for one packet to go from one gateway to another. This technique has been tested with a large number of standard network protocols and services.

Entities Protected: This technique aims to protect the discoverability and reachability of the VPN gateways between networks. The presumptions is that if an attacker cannot locate and reach a gateway before the IP hopping takes place, he will not be able to launch an effective attack against that gateway or the systems behind that gateway.

Deployment: This technique would be deployed on the VPN gateways in a network. Clients that are operating within this private network should not need to be modified.

Execution Overhead:

- None

Memory Overhead:

- None

Network Overhead:

- Changing the address information in packets may have an impact on the delivery times.

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☒ Reconnaissance

☒ Access

☐ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: The method for deriving and delivering the shared secret is secure.

Weaknesses: One weakness of this technique would be having an insufficiently large pool of IP addresses to use for hopping. If the pool is too small, an attacker could focus more on the limited addresses or be able to predict which addresses will be next with better accuracy. In addition, this would give an attacker with adequate resources the ability to launch DoS type attacks against the entire limited address space cutting off communication at that gateway. If it is possible for an address to be chosen twice or more in a row due to the random selection, it might give an attacker larger windows to mount an attack. If the systems that are part of the VPN are also part of a local network, it may be possible for an attacker to compromise a system within that local network then launch an attack on the systems that are part of the VPN directly. This technique is also not effective if an attacker is able to locate the target and mount an attack before the hopping takes place. If an attacker can analyze traffic, he may be able to use other aspects of the network traffic besides the address to determine where the current targets are located. In fact, the protection offered by this technique is only based on limited reachability. For example, this technique provides no protection against client-side attacks (e.g., browsing to a malicious website).

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique increases the amount of work an attacker has to do to discover the targets if he is using IP scans. An attacker would need to scan random IP addresses in order to discover target that is constantly changing addresses as opposed to scanning for a fixed host then mounting an attack. However, this technique does not provide any protection against an adversary that can reach a host using higher-level protocol information (web browsing or application-level communication).

Availability: This technique is being prototyped and tested by the Air Force Research Laboratory as a proof-of-concept but does not appear to be publicly available at this time.

A similar commercial product is available from Invicta: http://www.invictanetworks.net/. Another similar commercial product is available from Telecordia: http://www.telcordia.com.

Additional Considerations: The protection only focuses on masking IP addresses. Note that a host can be reached by many other means: browsing to a website, application-level communication, peer-to-peer (P2P) traffic, etc. The limitations imposed by the technique and the functionality impacts may outweigh the protection offered.

Proposed Research: This technique might offer more protection if it was implemented on individual systems as opposed to at the gateways. This would help protect against any scanning or attacks that are targeting the participants in the VPNs directly. Additional randomization of protocol fields or the inclusion of dummy traffic might also offer more protection for attackers that are able to perform traffic analysis. However, implementing at the level of individual hosts significantly increases the overhead.

Funding: Air Force Research Laboratory

5. DYNAMIC PLATFORMS

5.1 SECURITY AGILITY TOOLKIT

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Exploitation of Trust

Details: This technique [30] helps mitigate the damage that can be done on a system by restricting the access an application or process currently holds in the event of attack detection. It can restrict high-level access like read/write permissions to a file as well as low-level access such as system calls. It also has the ability to restrict external connections.

Description:

Details: This technique provides a toolkit to wrap around executables. It allows the injection of greater access control mechanisms with the ability to change them during program runtime. The toolkit is meant to supplement general intrusion detection system (IDS) frameworks. The idea is that if a detection of a certain threat or activity is encountered, the dynamic security policy of the affected applications can be dynamically changed. It could increase auditing, isolate affected processes, or even take measures like killing certain programs. There is an Agility Authority on each host that manages the agile processes for that host. Above that, an Agility Authority Manager distributes policy updates to each Agility Authority. The IDS can either send response directives directly to each Agility Authority or to the Agility Authority Manager. After a response directive is received, the policy is adjusted accordingly and actions are taken according to those new policies to mitigate the threat.

Entities Protected: This technique protects the OS when suspicious activity or threats are detected.

Deployment: This technique could be implemented into the OS at the kernel level to enable functionality to wrap around existing executables.

Execution Overhead:

- Will incur some unknown overhead while checking for policy updates and applying policy checks.

Memory Overhead:

- Will incur some unknown overhead by injecting the policy code into the running program.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☒ Persistence

Interdependencies: This technique relies on a framework that includes IDS, event analyzers, and response units to trigger policy changes. If the attack cannot be detected, this framework does not work.

Weaknesses: A potential weakness is the reliance on a separate detection mechanism. A stealthy attacker could avoid detection and carry out their attack without extra hindrance. An attacker could also potentially use the policies to cause a denial of service to the system by intentionally triggering the strict policies. This technique does not provide any protection against the first attack. It can only adjust the policy afterwards.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: Depending on how this was implemented, it may have some impact on the attackers. If the policies were implemented in a strict fashion as the starting policy, it could limit what the attacker could do to a system after compromising an application. If the attacker is detected, it could make it more difficult accomplish their task if they had not completed it before being detected.

Availability: This technique was prototyped by the authors but is not publicly released.

Additional Considerations: This work lacks many specifics. The technique relies on an external detection mechanism, so the effectiveness of this technique relies on the effectiveness of that mechanism. Since there is no perfect detection mechanism, this could significantly decrease the effectiveness of this technique. Also, the technique does not provide any protection against the first attack. It only relies on limiting damage afterwards. Moreover, policy changes can break applications and functionality.

Proposed Research: A possible future direction for this technique would be to make it more integrated with a detection mechanism. This would make the technique less reliant on external triggering mechanisms. Combining this technique with other movement techniques would increase the overall effectiveness of this method. If other techniques or guards are able to detect more specific types of attacks, that could be implemented as another triggering source for this technique. It is also crucial to understand the impact of policy changes and their effectiveness in stopping an attack.

Funding: DARPA

5.2 GENESIS

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [31] defends against different threats depending on how it is implemented. If it is implemented with ISR, it can defend against code injection attacks. If it is implemented with calling sequence diversity (CSD), it can partially mitigate attacks that divert the control flow of a program. It can protect against attacks that target function calls that exist before the program is loaded but not function calls that are generated during runtime.

Description:

Details: This technique involves applying runtime software transformations to a program. The program is run in an application-level VM called Strata. Strata with software dynamic translation can change a program by injecting new code, modifying existing code, or controlling program execution in some manner. Strata examines and translates all program instructions before they execute on the host processor. Two transformation methods were prototyped to test this technique.

The first method involves CSD. The method involves modifying the compiler to insert annotations into the code whenever there is a static control flow switch. It will XOR three keys each time this switch is made and will be compared to an expected key to verify it was a valid switch. The first key is generated at load time and is not accessible by or stored in the running program. The second and third keys are the source and destination keys. If an unexpected jump or control flow diversion is interested, Strata will dynamically generate the key check.

The second method involves modifying the linker to allow Strata to use ISR. This method uses 128-bit AES encryption instead of XOR. The linker marks all application and library code as encryptable, appends a tag to each instruction, and adds padding as necessary to properly align the blocks for AES encryption. Strata will encrypt the application when it loads and decrypts the instructions as they are needed for execution. It will then check the instructions for the proper tag. If the instruction is valid, it will remove the tag and add it to a cache to decrease decryption costs.

Entities Protected: This technique helps protect the OS by making applications more difficult to exploit.

Deployment: The Strata VM is deployed as a standalone application on the system and does not require modifying the OS. In order to use the methods of diversity discussed in the report, the compiler and linker on the system would also need to be modified to support each diversification method.

Execution Overhead:

- The ISR method adds about a 17% increase in overhead against the SPEC CPU2000 Benchmarks.
- The CSD method adds an average 54% increase in overhead against the SPEC CPU2000 Benchmarks.
- The emulation layer (Strata) can impose significant execution overhead.

Memory Overhead:

- Some additional memory will be required for the Strata VM and any keys or instructions it needs to store.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This technique relies on an emulation layer (Strata).

Weaknesses: This technique relies on the security and integrity of the VM. It assumes there is no way for an attacker to disable protections on the VM's memory sections and the VM implementation is sufficiently bug-free. The authors claim to protect the system calls that could disable these protections but a method may exist to disable those protections indirectly. In addition, this technique does not provide any protection against ROP attacks. This technique is also weak against memory secrecy violations.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: Both methods used in this technique will increase the amount of work needed to exploit the application. In both cases, more advanced control injection attacks, such as ROP, could be used to bypass these protections.

Availability: This technique was prototyped by the authors but is not publicly released.

Additional Considerations: The ISR method breaks self-modifying code and just-in-time compilers (e.g., Java). Running every application on top of an emulation layer can have significant execution overhead, which makes this technique impractical.

Proposed Research: A future research direction for this technique might be investigating methods to increase the protection provided by the CSD method. A larger direction might be combining this technique with an N-version programming technique. This would increase the overall difficulty in exploiting the application because now the attacker has to break multiple diversifications with one input. An efficient protection against code injection and ROP is an open problem.

Funding: DARPA

5.3 MULTIVARIANT EXECUTION

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This technique [32] combats code injection attacks by having each running variant use a different system call mapping and unpredictable stack direction. Each variant uses the same input making it difficult to inject code that will work with all mappings simultaneously. The stack direction change will result in a different flow of instructions in the program and libraries as well as providing different library entry points between the variants.

Description:

Details: This technique involves running multiple variations of the same program. A separate monitoring program monitors all variations. The level of monitoring can vary from each program having the same result down to checking each instruction executed. This technique focuses on synchronizing all variants at the system call level and each variant should make the exact same system calls. If any inconsistency is detected, all variants are terminated and restarted. The monitor is implemented as a user-space, unprivileged program. It monitors the arguments of system calls and all communication between the variants as well as interactions with the kernel.

There are some system calls that must be executed by the monitor on behalf of the variants to keep them synchronized. These would include system calls that change the state of the system or return volatile results. In this case, the results of the system call are passed to the variants. Variants are automatically generated by modifying the stack growth direction and system call number mapping but the technique is capable of any variation technique as long as the system call invocations are the same.

Entities Protected: This technique protects the OS by making the exploitation of an application more difficult.

Deployment: The monitor and variants are implemented as standalone applications running on a system. The variants are generated by using a modified compiler and modified system libraries.

Execution Overhead:

- Number of variants + monitor overhead.
- Additional time added for variant synchronization and communications.
- Average monitor overhead of ~17% with two variants.
- Average monitor overhead of ~30% with three variants.
- Average monitor overhead of ~37% with four variants.

Memory Overhead:

- Number of variants + monitor overhead.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: This technique requires a good source of randomness for the system call number randomization. This technique will not work with variants that cause a program to produce differing sequences of system calls.

Weaknesses: The actual dependency of attacks on the variants is unknown. This technique does not stop attacks against higher-level protocols. The multivariant monitor can be compromised specifically as it is in the “untrusted” zone. For example, the monitor can falsely indicate that the variants agree on a result. In addition, this technique is focused on integrity attacks and it does not protect against data leakage (exfiltration) attacks against one of the variants. The granularity of detection is also limited to system calls. Modifications to the user space code that keeps system calls intact remain undetected.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: This technique increases the difficulty of exploiting an application. It requires the attacker to provide input to a program that will simultaneously break all running variants. The impact will be different depending on the diversification method used on the variants. Some diversification techniques include stack reversal, instruction set randomization, heap layout randomization, stack base randomization, variable reordering, system call number randomization, register randomization, library entry point randomization, stack frame padding, code sequence randomization, equivalent instructions, program base address randomization, program section reordering, and program function reordering.

Availability: This technique was prototyped by the authors but is not publicly released.

Additional Considerations: The technique has significant overhead, as it requires many variants. It, however, does not protect against the compromise of one variant. The actual impact of diversity on the attacks is unknown. This technique also lacks many important specifics (types of diversity applied and its impact).

Proposed Research: It may be possible to compose some techniques while still preserving the required properties of this technique. It would be worthwhile to explore which methods can be composed together in a manner that does not cause unintentional divergences or false detections. More specifics are needed for a technique like this.

Funding: Air Force Research Laboratory

5.4 DIVERSITY THROUGH MACHINE DESCRIPTIONS

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Technique Mitigated: Code Injection

Details: This technique [33] is meant to mitigate mass code injection attacks. Each system would potentially need their own custom exploit to work because of all the varying system modifications and configurations.

Description:

Details: This technique involves using a VM and compiler machine descriptions to create a diverse set of architectures. This will regenerate all the machine-dependent and architecture dependent parts of a complete OS. Various items can be changed in these machine descriptions including the following:

- differing size of operations with different instruction sets and instruction encoding
- different number of registers
- different machine byte and word sizes
- different endiannesses
- different representation of signed integers
- different stack directions
- using one or multiple stacks
- different calling conventions such as alignment, ordering, padding, registerization, stack adjustment, and return value handing
- alignment padding in stack frames and data structures

The kernel would be able to have changes such as different sizes of standard types and linker relocation codes. These machine descriptions could be randomly generated. These architectures could be periodically applied to one machine or across many machines.

Entities Protected: This technique protects the OS as a whole.

Deployment: This technique is deployed inside of a VM and is composed of an entire system.

Execution Overhead:

- Some overhead imposed by running inside a VM.

Memory Overhead:

- None

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

- ☐ Data
- ☐ Source Code
- ☒ Compiler/Linker
- ☒ Operating System
- ☐ Hardware
- ☐ Infrastructure

Expertise Required to Implement:

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☐ Custom Programmer (General Knowledge)
- ☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

- ☒ Seamless
- ☐ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This technique relies on a virtualization layer. It also require a diversification layer to create the variants.

Weaknesses: The technique does not protect against application-level attacks. It does not protect against ROP attacks either. The virtualization layer is also a single point of failure and can be attacked. In addition, the technique does not provide any protection against targeted attacks on one platform.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: This technique would increase the amount of work an attacker would have to do to attack a large number of systems. An attacker may still be able to leverage non-code injection attacks against these systems that work across multiple architectures.

Availability: This is a research idea proposed by the author and has not been implemented.

Additional Considerations: This technique is theoretical and lacks many specifics. Constant recompilation of the entire OS could impose a large operational overhead. In addition, changing so many aspects of a system could potentially have unforeseen adverse effects on applications and can break functionality. Implementing this technique can be very difficult. See a discussion on diversity in [68, 69].

Proposed Research: A possible research direction for this technique might be coming up with a way to better automate this process. It may also be worthwhile to investigate potential side effects of changing so many parts of the architecture. This idea might be able to be expanded to mix in different versions of libraries and base OSs as well.

If a technique like this could be reasonably automated (see [70]) and any side effects of architecture randomization explored, a larger future direction could be combining this with a technique like Trusted dynAmic Logical hEterogeNeity sysTem (TALENT). This would allow for a large space of platforms to be dynamically generated or periodically regenerated.

The impact of different types of diversity on attacks has to be studied.

Funding: Unknown

5.5 N-VARIANT SYSTEMS

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [34] can be implemented with different application variants to target specific threats. The instruction set tagging variant gives each running variant their own instruction set. Since each variant is passed the same input, this will help mitigate code injection attacks because the attack might succeed on one variant but would presumably fail on another. The monitor would catch this divergence and restart the variants. Address space partitioning is the second type of variant used in this technique. This variant will help mitigate control injection attacks because each variant is mapped to a separate location in memory. This will only help mitigate control flow attacks that rely on fixed addresses. Attacks that know the relative location of their target can still succeed.

Description:

Details: The idea behind this technique is to run multiple variants of the same application simultaneously without relying on anything to be secret. It contains a polygrapher, the application variants, and the monitor. The polygrapher takes input and passes it to all the variants. The monitor watches the variants for a divergence and, if one occurs, restarts all the variants in a known good state.

This technique relies on a couple properties to work correctly. The first property is a normal equivalence that says when a variant is in a normal state, that state should be equivalent to a state in the unmodified, original process. The second property is a detection property that says certain attacks should be detected as long as the normal equivalence is satisfied. If a variant enters a compromised state then another variant should enter an alarm state or anomalous state that is detectable by the monitor.

The proof-of-concept was built into the Linux kernel and tested with two types of variants. The monitor synchronizes the variants at the system call level. System call wrappers are created so system calls can be shared between variants. System calls are broken into three categories: shared, reflective, and dangerous. Shared system calls interact with external state, reflective system calls observe or modify properties of a process, and dangerous system calls can break the assumptions of the technique.

The first variant tested was the address space partitioning. This utilizes the linker to load the data and code sections of the program at sufficiently different addresses while ensuring they will not overlap. The second variant tested was instruction set tagging. This utilizes a binary rewriter to insert a tag into each instruction and software dynamic translators to interpret these instructions during execution. Each variant would use different tags.

Entities Protected: This technique protects the OS by making the exploitation of an application more difficult.

Deployment: This technique is built into the OS and wrappers are created for some system calls.

Execution Overhead:

- N times slow down for N variants plus additional overheads as follows.
- 2 Variants with Address Partitioning: Unsaturated Server: 17.6% Increase, Saturated Server: 48% Increase.
- 2 Variants with Instruction Tagging: Unsaturated Server: 28% Increase, Saturated Server: 37% Increase.
- CPU-bound services will have a high overhead because each variant will duplicate computations.

Memory Overhead:

- Number of variants + monitor + polygrapher overhead.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This technique can only be used with diversification techniques that use variants similar enough to satisfy the normal equivalence property. It also relies on separate diversification techniques.

Weaknesses: This technique does not protect against application-level attacks. Also, the monitor can be a single point of failure. In addition, the technique does not provide any protection against ROP attacks and memory secrecy violations. It does not provide any protection against data leakage attacks on one variant either.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique increases the difficulty of exploiting an application. It requires the attacker to provide input to a program that will simultaneously break all running variants. The impact will be different depending on the diversification method used on the variants.

Availability: This technique has been implemented by the authors and is available at http://www.nvariant.org.

Additional Considerations: The technique lacks many specifics and the actual diversification techniques and their impacts are unknown. Also the technique kills programs that use the execve (execute program) or unrestricted mmap (map file into memory) system calls. Operating System Signals may cause the variants to diverge because variants might be in slightly different states when they receive the signal (this restricts the functionality of the technique). Variants using user-level threading may cause false detections because of the difference in thread interleaving causing different sequences of system calls. This could also potentially allow an attacker to exploit race conditions. Various non-attack inputs can cause false detections making this prototype less feasible for real services. In addition, running many variants may be impractical.

Proposed Research: A future direction for this technique would be to explore additional variant diversification methods. This technique could also be enhanced to work with more system calls and OS components like other similar techniques. The impact of diversification techniques on attacks must also be studied.

On a larger scale, it may also be possible to compose some techniques while still preserving the required properties of this technique. It would be worthwhile to explore the space of diversification methods and determining which methods can be composed together in a manner that does not cause unintentional divergences or false detections. Some diversification techniques include stack reversal, instruction set randomization, heap layout randomization, stack base randomization, variable reordering, system call number randomization, register randomization, library entry point randomization, stack frame padding, code sequence randomization, equivalent instructions, program base address randomization, program section reordering and program function reordering.

Funding: DARPA, National Science Foundation

5.6 TRUSTED DYNAMIC LOGICAL HETEROGENEITY SYSTEM

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection, Control Injection, Scanning, and Supply Chain

Details: This technique [35] can help mitigate a OS and architecture dependent attacks. Since the application is migrating between systems with different libraries, architectures, and layouts, it is more difficult to construct exploits that will work under every platform. The attacker also may not be able to predict when the application will migrate or which platform the application is currently running on. The fact that the application can be constantly moving makes passively scanning and collecting information less useful. Changing hardware platforms also makes supply chain attacks more difficult.

Description:

Details: The Trusted dynAmic Logical hEterogeNeity sysTem (TALENT) is a technique that involves making a running application migrate between different platforms (OS and CPU architecture) while preserving the state of that application. This state can include any files the program was using or sockets the program had open. These platforms have hosts with virtual containers. Each can be implemented with a different OS, different hardware, a different architecture, and different versions of libraries. The application being preserved is precompiled for each platform. TALENT needs compiler support to create checkpoints and containers to preserve the environment.

The current implementation uses Linux and BSD platforms. A centralized controller manages the migrations. The migrations can currently trigger at random intervals or via manual interaction.

Entities Protected: This technique protects the OS and applications running on it by continually shifting the attack surface.

Deployment: This technique is deployed across multiple systems. A special compiler allows a program to be periodically checkpointed. The source code of the program needs to be modified to be able to support the checkpointing.

Execution Overhead:

- Minimal overhead imposed by the checkpointing mechanism.
- A few seconds of downtime during migration.

Memory Overhead:

- None

Network Overhead:

- The state of the programs will be transferred between machines.
- Control messages passed between the platforms.

Hardware Costs:

- A system capable of a virtual infrastructure or additional machines to host each platform.

Modification Costs:

☐ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☒ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This technique relies on a detection mechanism for effective jumping if it is not using a random jumping scheme.

Weaknesses: There are a couple ways this technique could be less effective. If the platforms do not migrate fast enough, the attacker may be able to get an exploit together and attack the current machine. The technique does not provide any protection against higher-level protocol attacks. It also does not protect against attacks on one machine.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique would slow down an attack and make it more difficult for them to exploit the machines but it would not stop them completely. An attacker still has the chance of exploiting a system and achieving their goal before the application migrates. An attacker could also try to leverage more advanced Control Injection attacks that could work across numerous systems.

Availability: This technique has been prototyped by the authors and is available as a government off-the-shelf (GOTS) product.

Additional Considerations: Using this technique on every application can impose a very high overhead. It must be used to protect only a selected set of important applications.

Proposed Research: There are a number of future directions this technique could investigate. One direction would be implementing a recovery mechanism. This would allow the technique to clean up and recover from attacks. Another future direction would be adding data integrity checks. Currently, the technique has no integrity guarantees. Finally, another direction would be creating a distributed command and control mechanism to eliminate the single point of failure.

A possible direction in the future may be to combine this technique with a cloud concept to have a large and dynamic set of platforms to choose from at all times. This would make it less predictable which platforms would be in the migration sequence.

The impact of OS and architecture diversity on attacks must also be studied.

Funding: Air Force

5.7 INTRUSION TOLERANCE FOR MISSION-CRITICAL SERVICES

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Technique Mitigated: Resource

Details: This technique [36] combats resource attacks such as DoS and data integrity attacks. It mitigates the impact of DoS attacks by trying to ensure there are enough resources on a platform to run the service. It will terminate noncritical services to free up resources. If not enough resources can be freed, it will change to a different platform. This technique mitigates data integrity attacks by implementing a voting scheme for the service results.

Description:

Details: This technique aims to make critical web services more survivable in the face of attack. This is composed of a frontend that accepts requests from clients, some number of diverse platforms serving the same service, and a surveillance node that monitors the platforms and deals with voting. The platforms can have different OSs and different web servers to vary their attack surface.

Each platform implements a resource reallocation method that monitors the system. It is composed of a resource reallocation manager, health monitor thread, and survivability evaluation thread. The health monitor collects performance information from the OS and forwards that information to the survivability thread. This thread determines if resources need to be changed based on the performance. If resources need to be adjusted, the resource manager will start to terminate non-critical services to free up additional resources. If not enough resources can be freed to ensure acceptable performance, the platform is taken offline for recovery.

Each platform also has a result acceptance tester. This component tests the logical reasonableness of the result and the execution time required to obtain that result. The platforms can operate in two modes. There is an active mode where a set of active nodes process a request simultaneously and the result is voted on and processed by the surveillance node. There is also a passive mode where only one platform is active. If that platform does not pass the acceptance test, it is replaced by another platform and recovery is performed on it.

Entities Protected: This technique protects specific applications and services to ensure they continue to operate under attack.

Deployment: This technique would be deployed in the overall network infrastructure.

Execution Overhead:

- Up to an additional 50% time may be needed to process a request.

Memory Overhead:

- None

Network Overhead:

- Additional out-of-band network needed for surveillance.

Hardware Costs:

- Additional platforms to host the additional variants.
- Additional network infrastructure to support this platform configuration.

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☒ Persistence

Interdependencies: The system relies on a detection capability that monitors the resources.

Weaknesses: This technique assumes that only one active platform will be compromised at any given execution cycle. The voting mechanism would see this as a valid result and it could compromise the integrity of this technique. Also this technique does not stop any attack. It just tries to mitigate DoS attacks by resource management. In addition, data leakage attacks or low-observable attacks are not mitigated.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: This technique would make it more difficult for an attacker that is trying to deny service. This technique might not slow down an attacker with a different goal. An attacker could still carry out other attacks that take advantage of the voting system or exploit the system at a lower level.

Availability: This technique was prototyped by the authors but is not publicly available

Additional Considerations: This technique provides no additional protection. It just mitigates the impact of certain types of attacks. Also the technique lacks specifics. For example, it is unclear what types of resources are monitored. What happens to obscure resources that can run out (e.g., file descriptors or certain ID numbers)? A similar technique is proposed in [67].

Proposed Research: A possible enhancement to this technique would be to make the voting system more difficult to bypass. Currently, only two results need to match for a result to be accepted. This could be expanded to more systems.

A larger direction for this technique would be to combine this technique with other movement techniques on the platforms. This would increase the diversity between each platform and make the platforms resistant to a larger set of attacks.

Also a study must be conducted to enumerate all possible resources that can be attacked in a machine during a DoS attack.

Funding: University IT Research Center Project, University of Incheon, Korea Information Security Agency Research Project

5.8 GENERIC INTRUSION-TOLERANT ARCHITECTURES FOR WEB SERVERS

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection, Control Injection, and Scanning

Details: This technique [37] helps reduce the attack surface [45] of the services by not making them directly accessible from the outside, limiting the types of traffic that can reach it, and running on multiple diverse systems. Each request can use a different subset of diversified servers and the results are voted on making it more difficult for one attack to be universal. The servers are also hidden behind the firewall and proxies making information collection attacks more difficult as well.

Description:

Details: This technique aims to be a system capable of diagnosing issues, repairing itself, and reconfiguring itself in order to continue to provide a service in the event of attack. It consists of a firewall in the front that filters all traffic except http traffic. The traffic that gets through is fed to a network of proxy servers. These proxy servers communicate with the web servers. Each web server is a diversified system containing various architectures, OSs, and software packages while providing the same content. The proxy systems are diversified in a similar manner but have been hardened to be further resistant to attack.

The proxy servers have the ability to take on differing roles. The proxy servers choose a leader and this leader handles all requests from the firewall. It is responsible for determining which subset of web servers should be used to process the client request. A different number of web servers can be chosen to process the request based upon how critical the service is or the current alert level of the system. It will also be responsible for making sure everything is load balanced so some servers are not overworked. One of the proxy servers is also chosen as an adjudicator that manages connections to the shared database if one is needed. It has the ability to filter out any suspicious looking SQL queries. Each proxy is capable of taking on one of these roles if something happens to the existing elected proxy. Each proxy also has an alert manager on it. Each proxy and server can be in a trustworthy state, a suspected state, or a corrupted state. The alert managers help decide what action should be taken in the event of an alert from any detection mechanism (described later). When something receives an alert, a vote is taken amongst the proxies to verify alert. An action is then taken depending on the role of the corrupted component. Each proxy is in a different administration domain to prevent one administrator from taking over all proxies. The leader proxy also has the ability to filter out any suspicious looking http requests.

This technique incorporates a number of different detection mechanisms. The first is an agreement protocol. This is a voting mechanism to determine if a server was corrupted. Each server processing the request sends a cryptographic hash of the response excluding the header back to the current leader. The majority response is then used to be sent back to the client. The proxies use this same voting protocol when an anomaly is detected in any of the systems to come to a consensus on a countermeasure. The adjudicator proxy uses this protocol to verify SQL queries before executing them on the database as well.

The next detection mechanism used is a combination of network and host-based intrusion detection system. Host-based intrusion detection systems are placed on every host and a network-based intrusion system is used to monitor the traffic between servers and proxies. Each web server is rebooted periodically from a read-only trustworthy source.

In between each two reboots of a system, another detection mechanism is implemented. This is a challenge-response protocol implemented by each proxy. A proxy can periodically send out a challenge to any other proxy or server about a file on that system. The response is checked against a precomputed response. There must be enough challenges generated to last between two reboots of a system.

The final detection mechanism implemented is a runtime verification of the proxies. This checks the behavior of each proxy during its execution. The system is modeled by a finite-state machine and the state is monitored. There are different models depending on the current role of the proxy. The proxy can both monitor its own behavior as well as the other proxies' behavior.

Entities Protected: This technique is used to protect the availability and integrity of web services.

Deployment: This technique would be deployed in the overall network architecture.

Execution Overhead:

- Duplex and triplex regimes have 200% and 300% overhead plus additional overheads as follows.
- For one server without database access and a 1MB file, this added about a 31% overhead.
- For three servers without databases access and a 1MB file, this added about a 33% overhead.
- Database access time approximately doubled with this technique.

Memory Overhead:

- None

Network Overhead:

- Additional network traffic generated by the additional servers.

Hardware Costs:

- Additional platforms to host the additional variants.
- Additional network infrastructure to support this platform configuration.

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☒ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: A firewall able to effectively filter everything but HTTP traffic. The service being provided should produce the same results under normal conditions on all systems. The technique relies on a detection mechanism.

Weaknesses: An attacker could launch a large-scale attack that results in all the web servers rebooting due to detections causing a denial of service. In addition, this technique provides no protection against targeted attacks on one web server. Also, data leakage attacks are not protected.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique could significantly increase the workload of the attacker. The attacker would need to create an attack that would work on an unknown majority number of diversified systems. An attacker may be able to leverage a script injection if it is not detected by any of the detection mechanisms. The attacker also cannot directly access the servers making it more difficult to do reconnaissance on those systems.

Availability: This technique was prototyped by the authors but is not publicly released.

Additional Considerations: The report lacks specifics on the types of diversity or how that may impact security. It could be prohibitively expensive at large scale. The technique only handles HTTP traffic. This method is limited and can only be applied to specific a system.

Proposed Research: One possible direction to look into would be to blacklist requests that caused a server to go into a bad state. Future requests that are on the blacklist could be blocked at the proxy. This would prevent a continuous denial of service attack using the same attack request continually. It may also be possible to combine this technique with other movement methods on the web servers. This could make them more resistant to a larger set of attacks and offer more detection mechanisms.

Funding: SRI International, DARPA

5.9 SELF-CLEANSING INTRUSION TOLERANCE

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [38, 47, 90] does not detect any attacks but assumes the system is continually under attack. While this would not stop an attacker from injecting code, the minimal exposure time before cleaning a system would require a fast-acting exploit. This would also stop attackers from continuously persisting on these systems.

Description:

Details: The self-cleansing intrusion tolerance (SCIT) technique aims to decrease the exposure time of a system by rotating it with copies. The copies that are not being used are cleaned and restored to a pristine state. Each system copy is implemented in a virtual environment. There is a separate system with a network attached memory utility that stores persistent short-term information or session data between the systems. The final component is a controller that manages the rotation of the systems and how long each system copy is exposed. The systems can be in one of four states. The first state is *active* where it is online and accepting/handling requests. The second state is *grace period* where it stops accepting new requests and finishes processing existing requests. The third state is *inactive* where it is taken offline to be restored. The final state is *live spare* where the system has been restored and is ready to become active. There can be either one active server at a time serving one service or multiple active servers serving multiple services. The latter would require additional algorithms to determine which systems could be easily brought down next. The systems are rotated on the order of minutes. The systems are on their own private virtual network and are not directly accessible from the Internet. Connections to the systems are managed by a load balancing system.

Entities Protected: This technique protects servers by limiting their exposure time.

Deployment: This is a contained virtual environment and could be deployed on the servers.

Execution Overhead:

- Some unknown overhead due to virtualization.
- Significant overhead for cleaning the VMs.

Memory Overhead:

- None

Network Overhead:

- None—self-contained virtual network.

Hardware Costs:

- Additional hardware to support a virtualized environment.

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☒ Hardware

☒ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☐ Exploit Development

☐ Attack Launch

☒ Persistence

Interdependencies: The technique requires a virtualization infrastructure in place. It also requires an automated re-imaging capability.

Weaknesses: The networked memory does not have any protection or integrity control. Since it is accessible via all systems, an attacker could attempt to quickly corrupt or change the contents of this storage. Another weakness is that every system is the same. If the attacker can find a working exploit against one system, it would work on all systems at once. In fact, since exploits work very fast, this technique provides little protection. The system also does not protect against data leakage attacks.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☒ Disable Movement

Impact on Attackers: This technique does not stop an attacker from exploiting a system. It decreases the amount of time an attacker has to accomplish their goal. This makes it more difficult to persist in the network. An attacker would have to compromise the load balancer or the networked memory system or quickly jump to new systems as the older ones are being re-imaged.

Availability: This technique was prototyped by the authors but is not publicly released; see http://scitlabs.com/.

Additional Considerations: The technique provides little protection at a large cost. The attacker can always jump to the new platforms (since they are identical) and continue to persist. Moreover, the overhead to re-image the systems can be very large. The technique is also limited to specific servers that are almost stateless or the state can be persevered in the configuration (e.g., DNS server).

Proposed Research: One direction for this technique would be to develop a better way to protect the network memory. If an attacker can continually change or corrupt the data, the effectiveness of this technique is significantly decreased. Another direction this technique could take would be to introduce diversity into the OSs. Different architectures, OSs, and servers that provide the same function could be used to increase the workload of the attacker. This would reduce the likelihood of an attack working across all systems. Also preserving the state beyond configuration files is a direction to explore.

Funding: Lockheed Martin, Virginia Center for Innovative Technologies

5.10 GENETIC ALGORITHMS FOR COMPUTER CONFIGURATIONS

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Technique Mitigated: Scanning

Details: This technique [39] has a long-term security goal and does not actively defend against or respond to attacks. The idea is that the evolution of configurations over time effect the lifetime of exploits and the varying configurations amongst systems helps prevent exploits from working against multiple machines. The evolving configurations make collecting information about a specific machine less reliable.

Description:

Details: This technique aims to find more secure configurations of systems over time using ideas from genetics. The security of a configuration is defined as the number and severity of security incidents reported while that configuration was active. A configuration can consist of many parameters in a system such as which desktop manager is being used or which remote login protocol is being used. The ideas that are being used from genetics include selection, crossover, and mutation. Selection involves selecting the best configurations based on their security score. Crossover involves taking two configurations and combining elements of each one to create a new configuration. Mutation involves randomly changing parts of a configuration to make it different from configurations on other systems. The goal is to create configurations with temporal and spatial diversity. Temporal diversity means the configuration of one machine changes over time. Spatial diversity means multiple computers do not have the same configuration at a given time.

How this process works is that every system starts with the same configuration. Since no other configurations exist yet, it is mutated to create a new configuration. If the resulting configuration is reasonable, it is set as the active configuration. After some time has passed, the security score is calculated for that configuration. If the configuration pool is not full, this configuration is added into this pool otherwise it replaces the worst configuration in the pool. The next iteration would involve taking the two best configurations from the pool, doing a crossover to create a new configuration, and applying a mutation to add some additional randomness to it. This new configuration then goes through the same process of seeing if it is a reasonable configuration, making it active, and calculating its security score. This process repeats over many iterations until ideally there are configurations that have no security incidents.

Entities Protected: This techniques aims to protect the OS or servers by finding better configurations over time.

Deployment: This would be deployed on each system that has a similar purpose.

Execution Overhead:

- Unknown execution overhead due to reconfiguration.

Memory Overhead:

- None

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

- ☐ Data
- ☐ Source Code
- ☐ Compiler/Linker
- ☒ Operating System
- ☐ Hardware
- ☐ Infrastructure

Expertise Required to Implement:

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

- ☐ Seamless
- ☒ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

Kill Chain Phases:

- ☐ Reconnaissance
- ☐ Access
- ☐ Exploit Development

☐ Attack Launch

☒ Persistence

Interdependencies: A detection mechanism or set of mechanisms to be able to calculate the security score is critical for this technique. It also assumes that the entire security posture of a system can be controlled via configuration.

Weaknesses: This can be deceptive and give a false sense of security. A configuration can be chosen as good but it could be the case where the system was just not under attack at the time. Another large weakness of this technique is that the security score is based on detected attacks and relies on the systems being constantly attacked. A stealthy attacker could still carry out their attack. It may also be possible to manipulate the configuration selection by causing detections on configurations the attacker does not want. Moreover, many important security aspects of a system cannot be controlled with a configuration (see [66]). Also, it can take a long time to converge to a good configuration. The technique does not protect against data leakage attacks or one-time attacks either.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: Depending on the configuration options being changed, this may have little effect on the effort of an attacker. In ideal conditions and with strong detection mechanisms, it could affect an attacker over time by making it more difficult for them to perform an attack.

Availability: This technique was prototyped by the author but is not publicly released.

Additional Considerations: Frequently changing the configurations can be impractical. Functionality of the system may break because of reconfiguration. It can take a long time to converge to a good configuration. Also systems evolve over time. New software could be installed, old software removed, system patches applied, OS upgraded, etc. All these changes will result in new or removed configuration options. Also the system crucially relies on a perfect detection and monitoring capability to operate correctly.

Proposed Research: A possible future direction for this project would be to expand it from simple configurations to actual software as well. It might try running a service with a specific server then later try running the same service with a different server that provides similar functionality. This does not fix the reliance on detection mechanisms. Additional aspects that would need to be investigated are how the configurations are combined to create new configurations, how configurations are randomized, and how long a configuration would need to be active to get a reasonable security score. Combining this technique with other movement techniques that provide more proactive protection against classes of attacks might provide more overall protection. The impact of configuration changes on the functionality of the system must be studied.

Funding: Pacific Northwest National Laboratory

5.11 MOVING ATTACK SURFACE FOR WEB SERVICES

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection, Control Injection, and Scanning

Details: This technique [40, 92] can help mitigate a variety of attacks. Since the service is being served randomly between systems with different frameworks, libraries, architectures, virtualization technologies, and layouts, it is more difficult to construct exploits that will work under every platform. The attacker also may not be able to predict when the system will change or which system the load balancer will choose to service the request. The fact that the system is changing constantly makes passively scanning and collecting information less useful.

Description:

Details: This technique employed diversification at different levels of a system and across many systems to create a varying attack surface across all the systems. The technique is composed of a frontend load balancer, a pool of virtual diversified systems, and a trustworthy controller. The idea is to have a subset of the virtual systems online at a time and the load balancer chooses which one will service a request. The trustworthy controller manages the virtual systems. Virtual systems can be in one of three states: online, graceful shutdown, or offline. In graceful shutdown, they do not accept new request and finish processing any existing requests. In the offline mode, they are powered off. When a system is to come back online, it is restored to a pristine state. The virtual systems are randomly rotated in and out by the trustworthy controller. Three different mechanisms can trigger a system to be taken offline. The first mechanism can be event driven where anomalous events are detected or integrity checks fail. The second mechanism is a random selection where a system can be taken offline at any random time even if no anomalous activity is detected. The third mechanism is a maximum lifespan where a system is taken offline if the amount of time it has been online exceeds a limit in order to reduce exposure to attacks. This technique does not support services that require persistent state because they are wiped clean periodically. For this reason, the authors recommend a framework such a Representational State Transfer (REST) so the web servers can be stateless and still provide stateful services.

This diversification is done at the application level by using different implementations of the web framework being employed or different implementations of software that those web frameworks require. Diversification is done at the web server level by choosing different open source and commercial web servers. Additional diversification is achieved by hosting these web frameworks on the same web servers using different modules or technologies. Diversification is achieved at the OS level by choosing from a variety of open source and commercial OSs including Solaris, Windows, BSD flavors, and Linux flavors. Additional diversification is achieved by putting in a mix of 32-bit and 64-bit versions. Diversification is achieved at the virtualization level by using a mix of hypervisor-based and OS level virtualization technologies that support some form of checkpointing and restoration of systems. With all these levels of diversifications, the authors were able to come up with 1554 unique combinations.

Entities Protected: The primary function of this technique is to protect a web service, but the diversification also helps protect the OS as a whole.

Deployment: This technique would be deployed in the web services.

Execution Overhead:

- K replicas impose at least K times overhead in execution.
- Some overhead imposed by running in a virtual environment.

Memory Overhead:

- K times memory used.

Network Overhead:

- None

Hardware Costs:

- Hardware to support the various virtualization setups.

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☒ Infrastructure

Expertise Required to Implement:

- ☐ Simple Configuration/Installer
- ☐ Complex Configuration (System Admin)
- ☒ Custom Programmer (General Knowledge)
- ☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

- ☐ Seamless
- ☒ Simple Configuration
- ☐ Complex Configuration (System Admin)
- ☐ Expert Operator

Kill Chain Phases:

- ☒ Reconnaissance
- ☐ Access
- ☒ Exploit Development
- ☐ Attack Launch
- ☒ Persistence

Interdependencies: If relying on the event-driven rotation, it is necessary to have good anomaly and integrity checking mechanisms in place. This technique also requires a web framework that supports running stateful services on stateless servers if stateful services are required. This is not straightforward to achieve for arbitrary web services.

Weaknesses: This technique does not protect against web service logic bugs or failure to sanitize input. As a result, attacks like SQL injections could be leveraged if the service does not properly sanitize input or put other mitigations in place. The load balancer and trustworthy controller are both static machines and could be potential targets for an attacker. If an attacker can compromise the trustworthy controller, he could control or stop the system rotation process. If the system rotation process is not done quickly enough, the attacker may still be able to accomplish his goal if he is not trying to be persistent. It is also possible an attacker has a set of attacks that work only on certain combinations of software and the rotations of systems may eventually get to that configuration. More importantly, this technique does not protect against data leakage attacks or attacks against one machine.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☒ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique would slow down an attack and make it more difficult for him to exploit the machines, but it would not stop him completely. An attacker still has the chance of exploiting a system and achieving his goal before the active system changes. An attacker could also try to leverage more advanced Control Injection attacks that could work across multiple systems.

Availability: This technique was prototyped by the authors but is not publicly released. It is composed primarily of open source or commercial software.

Additional Considerations: The impact of diversity on identifying attacks is unknown. Having a large number of diversified systems would increase the management and maintenance complexity. This technique is only limited to a web server. Extending the technique to a generic service can be very difficult.

Proposed Research: This technique could potentially be combined with additional internal OS movement techniques to slow down the attackers further giving the system additional time to migrate systems. Ensuring that the trustworthy controller is isolated and the virtual systems are not able to manipulate it would also be a worthwhile avenue to explore. The impact of various types of diversity on attacks must also be studied.

Funding: Unknown

5.12 LIGHTWEIGHT PORTABLE SECURITY

Last Updated: 6/29/2012

Defense Category: Dynamic Platforms

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [41] helps mitigate persistent threats on a system by ensuring the OS boots into a clean and known-good state. The system can be rebooted in between sessions to return the system to a good state removing any infections incurred during the last session.

Description:

Details: The Lightweight Portable Security technique involves booting a system into an isolated and minimal OS. This OS resides only in the memory of the system and does not access any internal persistent storage devices. The OS is on a read-only bootable device or media ensuring that it cannot be corrupted or modified in a malicious manner. The OS is built off of the Linux OS and includes a basic set of applications such as a web browser, smart card middleware (such as the Department of Defense Common Access Card or U.S. Government Personal Identity Verification card), encryption software, file browser, image viewer, PDF viewer, text editor, remote desktop software, and SSH client. It also includes the ability to use external storage devices such as USB hard drives and memory sticks. The public editions of this technique allow a person to browse the Internet without putting their local machine at risk. The deluxe public edition has all the software of the regular public edition with the inclusion of additional software such as office software. The remote access version of this technique is meant to connect to enterprise networks and use internal network resources and it is customized for each particular customer.

Entities Protected: This technique protects a user session by booting into a known good and clean state. There are two primary use cases for this technique. If a person wants to browse untrusted websites and wants to protect his local computer, he can boot one of the public editions of this technique. A person might not trust the local computer and he wants to protect his online session. In this case, he can boot from one of the editions of this technique and do activities like online banking or connect to his work network securely without worrying about the local machine assuming the hardware/firmware is trustworthy.

Deployment: This technique would be deployed on a generic machine by booting from a read-only media.

Execution Overhead:

- Extra time required for re-booting into another OS.

Memory Overhead:

- Unknown memory overhead due to removal of hard drive.

Network Overhead:

- Some overhead incurred if connecting through trusted networks.

Hardware Costs:

- None

Modification Costs:

☐ Data

☐ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

(No modification is required)

Expertise Required to Implement:

☐ Simple Configuration/Installer

☒ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☐ Seamless

☐ Simple Configuration

☒ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☐ Exploit Development

☐ Attack Launch

☒ Persistence

Interdependencies: The technique assumes that secure and lightweight versions of the OSs are available.

Weaknesses: This technique does not protect against compromised hardware in the system or hardware connected externally to the system. An attacker could re-flash firmware on the machine and persist. It is also possible for an untrusted external hardware device, such as a USB hard drive or memory stick, connected to the local machine. Since the technique supports using external hardware, a new session may not be trusted if these external devices are mounted automatically. More importantly, the technique does not provide any protection after booting into a new OS. The sessions can still be compromised and information can still leak. The technique relies on rebooting after performing any important operation or for performing potentially dangerous actions (browsing an unknown website).

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique removes the persistence of attacks. It makes it harder for an attacker to remain on the system. Once a new session is initiated, any malicious code that was placed on the system will be removed.

Availability: This technique is available in three different editions. There is a public edition, public-deluxe edition, and remote access edition. The first two editions are free to download and use, but the third edition must be requested from the agency. See http://www.spi.dod.mil/lipose.htm.

Additional Considerations: This technique requires booting a system each time it needs to be used. It can have a very large overhead due to rebooting. In many cases, it is difficult to distinguish between benign and potentially dangerous actions. It must connect external devices for local persistent storage. The OS runs completely in memory so the host would need an adequate amount in the local machine.

Proposed Research: This technique does provide reasonable protection from persistent threats, but it does not address all the locations a persistent threat could reside. The hardware firmware inside of the host machine could have been tampered with in a malicious manner. An interesting research direction might be to see if it is possible to leverage trusted hardware technologies to verify hardware has not been tampered with as well. If the user intends to create a secure session because he does not trust the local machine, it would also be good to look into making sure potential untrusted or malicious external devices connected to the local machine are not automatically mounted into the trusted environment. One can also look into making reboots faster and more efficient.

Funding: Air Force Research Laboratory

6. DYNAMIC DATA

6.1 DATA DIVERSITY THROUGH FAULT TOLERANCE

Last Updated: 6/29/2012

Defense Category: Dynamic Data

Threat Model:

Attack Technique Mitigated: Resource

Details: This technique [42] was not designed to fight malicious input directly but it is more focused on unintentional faults. Since it reprocesses data and does voting on the results, it could help combat an attacker that is trying to manipulate or corrupt the output of a program or service.

Description:

Details: This technique aims to increase the fault tolerance of an application by reevaluating the input to a program using a different algorithm. These different algorithms can produce exact equivalent output or they could be general algorithms that produce approximations of the original output. The idea is a possible fault or corner-case for a specific input might be avoided if it is calculated in a slightly different or semantically equivalent fashion. The technique builds on the idea of N-version programming but uses a data-centric version of it the authors refer to as N-copy programming. Input is passed into independently developed versions of a program. The output of these is then passed to a voter that decides if the input is acceptable. If the output does not look acceptable, a new algorithm is chosen to process the input and the cycle is done again. If exact algorithms are being used, the voter can use the majority output as the good output. If it switches to more generic algorithms that produce approximations, then the voting can become subjective because the copies could produce different results that are still acceptable.

Entities Protected: This technique aims to protect a program by ensuring the output is acceptable.

Deployment: This would be implemented into the code of a program on a system.

Execution Overhead:

- There may be some additional processing overhead imposed if the program needs to reprocess the input.

- Running multiple copies of a program and waiting for voting results will add additional overhead.

Memory Overhead:

- Extra memory used by running multiple versions of the program (roughly N times for N copies).

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☒ Data

☒ Source Code

☐ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☒ Custom Programmer (General Knowledge)

☐ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: The technique assumes that N different (independent) version of the algorithm can be built.

Weaknesses: This technique relies on voting, so it is still possible for an attacker to corrupt all or the majority of the processes in order to bypass the added protection. It may also still be possible for an attacker to create output that still looks valid to the output checker so it is not rerun again with different algorithms. Another possibility is that the differing algorithms might have no effect on the malicious input.

Types of Weaknesses:

☒ Overcome Movement ☐ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: If an attacker is attempting to corrupt or manipulate the output of a program, this could make it more difficult. The technique is mainly focused on integrity protection. If the algorithms used to process the input are not sufficiently different between retries, an attacker may still be able to complete their objective. If an attacker can accomplish their goals without needing the programs to output, this may not have much of an impact.

Availability: This technique was tested by the authors but does not appear to be publicly available.

Additional Considerations: Creating a component that can accurately detect the validity of output could be difficult for a program or service with varying and dynamic output. Also developing N different algorithms is time consuming and requires redevelopment of an application.

Proposed Research: Some problems would need to be solved to make this technique more reasonable. One of those is coming up with a reliable way to determine if output of a program is valid. There are many applications and services now that have very dynamic and varying outputs so it may not be trivial to determine if output is valid. The same can be true for the varying input processing algorithms. It may not be an easy task to develop multiple ways to process the input inside of the application. It may also not be an option to use more approximate methods if the accuracy of the output is important. Automating the diversification of an algorithm is an important future direction.

Funding: NASA

6.2 REDUNDANT DATA DIVERSITY

Last Updated: 6/29/2012

Defense Category: Dynamic Data

Threat Model:

Attack Techniques Mitigated: Resource and Code Injection

Details: This technique [43] aims to help mitigate attacks that target specific data inside of an application by way of malicious input. Each variant is running a different transformation of the data such that one input would not be able to change all variants. This would cause a divergence and it would be detected by the variant monitor. The change can also be done at a lower level separating the address space of each variant or running each variant with different instructions. This helps mitigate some code injection attacks or injection attacks that rely on specific memory addresses.

Description:

Details: This technique is a variation of the N-variant programming technique. In involves running multiple copies of a program that each run transformations of the original data being protected without having to rely on secrets. These transformations should be semantically equivalent and reversible. A monitor can watch the values of the data in each variant to detect if there is a divergence and take appropriate action. This can be implemented on different levels such as having variants use different memory address spaces, different instructions, or different data representations.

The specific method for this technique analyzed was using different data representations. This was implemented to protect user identification (UID) and group identification (GID) that are used for determining permissions. This is implemented into the system kernel, then new system calls are created to allow for synchronization, and other system calls are modified accordingly to support the modified data. Each variant is modified to use new system calls for synchronization and to support the new UID and GID representations. The variants synchronize on system calls. Whenever one variant reaches a system call, it waits for the other to reach it as well. The inputs to the system calls are verified before execution. The system call is only executed once and the results are passed to each variant if it was an I/O based system call. If the program uses external file, such as configuration files, a new one is created and tailored toward the specific variant. If the program used the password file on the system and it contained some of the data being randomized, a new password file would need to be created for each variant.

Entities Protected: This technique aims to protect data entities inside of a running program on systems.

Deployment: Depending on the types of data being protected, it could be deployed at different levels. The implementation described is implemented into the OS kernel.

Execution Overhead:

- Running the Apache Web Server unsaturated with 2-Variant UID imposed a 13% throughput overhead and 14% latency overhead.
- Running the Apache Web Server saturated with 2-Variant UID imposed a 58% throughput overhead and 135% latency overhead.

Memory Overhead:

- There will be additional memory used by running multiple variants simultaneously.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☒ Data

☒ Source Code

☐ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: This should not be combined with diversity techniques that change the behavior of the program or make it perform semantically different. Such a technique would cause a divergence that would trigger a detection in their monitor.

Weaknesses: An attacker could still target data parts of an application that are not randomized if they can be used to mount an attack. An attacker could also try to use advanced control injection attacks that could still potentially affect many or all variants. Also the technique proposed is very limited in scope (only a very small portion of data on the system is randomized).

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: This could make it much more difficult for an attacker to corrupt certain important parts of an application if it was being protected properly by this technique.

Availability: This technique was prototyped by the authors but was not publicly released.

Additional Considerations: Techniques like this would have a larger overhead for computation-intensive programs. Each variant would have to do the expensive computations. In addition, it can be very challenging to expand this technique to the majority of data being processed on the system. There was no mention of recovery if the monitor detects something malicious.

Proposed Research: This technique was currently only implemented to protect data inside of the application. It could be extended to include some of the lower-level diversification techniques also described in the report, such as instruction set tagging and address space separation. This would make the application more resistant to different code injection attacks but would also add additional execution overhead as well. Additional research may also be needed to overcome possible false positive detections due to accidental divergences. These could happen because of operating signals reaching variants in different positions of execution.

Funding: National Science Foundation

6.3 DATA RANDOMIZATION

Last Updated: 6/29/2012

Defense Category: Dynamic Data

Threat Model:

Attack Techniques Mitigated: Code Injection and Control Injection

Details: This technique [44] helps protect against code injection attacks by randomizing any code injected into the program. All data that is written to memory within a certain class is randomized with a random key. This also helps protect against attacks that target pointers in general such as function pointers or return addresses. These would also be randomized by this technique using different keys. In addition, this technique would also provide some protection from attacks that attempt to read or write arbitrary memory locations. Any functions that attempted to write something would have that randomized as it was put into memory and reading arbitrary memory locations would result in that data being randomized with the key.

Description:

Details: This is a compiler-based technique that provides probabilistic protection by randomizing all the data that it stores in memory. All operands in a program within a class that read and write memory are instrumented to perform an XOR of the data with a random key. All operands that reference the same objects are grouped together. Each of these groups is randomized with a different key that is generated when the program is started. These groups are found during compile-time by using static analysis within the compiler. To improve performance, operands that are classified as safe are not instrumented. An operand is considered safe if runtime access to that operand can never violate memory safety. The compiler will then insert instructions that perform the XOR operations for reading and writing to memory in the appropriate locations. This technique also supports libraries. Wrappers can be created for the library functions and system calls that receive or return pointers.

Entities Protected: This technique protects the data applications store in memory.

Deployment: This technique would be implemented in a compiler on a system. Each program that wanted to use this technique would need to be compiled with this new compiler.

Execution Overhead:

- The average overhead for the tested benchmarks was 11% but it can be a wide range in either direction.

Memory Overhead:

- The tested benchmarks had an average memory overhead of 1%.

Network Overhead:

- None

Hardware Costs:

- None

Modification Costs:

☐ Data

☒ Source Code

☒ Compiler/Linker

☐ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☐ Attack Launch

☐ Persistence

Interdependencies: A good source of randomness for the key generation.

Weaknesses: It is still possible for an attacker to guess the randomization key to be able to read/write data to/from memory (technique assumes memory secrecy). It is also still possible to attempt to brute force the desired keys. This could result in a large number of program failures that would increase the probability of detection. An attacker may also be able to get to the desired memory object if there is a vulnerability in the same group of operands since they would use the same key. In order for this to be effective, it requires that all libraries also be protected via wrappers. If any libraries are overlooked, that opens the possibility to bypass this technique.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☐ Limit Movement ☐ Disable Movement

Impact on Attackers: This technique would make exploiting a protected application more difficult. An attacker would have to find a method to either leak the keys or guess the keys used to randomize the data.

Availability: This technique was prototyped by the authors but was not publicly released.

Additional Considerations: It requires program recompilation. The size of the programs increased by averages between 15% and 30%. Applying this technique to a wide range of programs can make it impractical.

Proposed Research: One larger direction this technique could take would be to combine with other memory protection techniques such as address space randomization. This would put further burden on the attacker that is trying to execute low-level attacks. Also it would be important to study what types of attacks can be mounted without crossing the groups (classes).

Funding: Microsoft Research

6.4 END-TO-END SOFTWARE DIVERSIFICATION

Last Updated: 6/29/2012

Defense Categories: Dynamic Data, Dynamic Software

Threat Model:

Attack Techniques Mitigated: Code Injection and Exploitation of Authentication

Details: This technique [45] has the potential to defend against different levels of code injection as well as some authentication attacks. Randomizing the instruction sets, script Application Programming Interface (API) randomization, randomizing the reference names of stored data, and randomizing components of code can help fight high-level code injection attacks like SQL injection attacks as well as low-level code injection attacks that target the internal application. They can also help fight attacks that compromise authentication like cross-site scripting (XSS) attacks that might try to inject code at a high level. Other diversification methods that can be used with this technique can help mitigate injection at additional levels.

Description:

Details: The idea of this technique is to compose many different randomization methods and apply them to aspects of a service that does not affect the functionality of the program. This would involve building functionality into the core or subsystems of a service that allows various aspects to be randomized. The example in the report is diversifying an Internet service. Some of the proposed diversification methods include changing Hypertext Transport Protocol (HTTP) keywords/syntax/headers/content encoding, Hypertext Markup Language (HTML) Document Object Model (DOM) structures/identifiers, SQL keywords/syntax, database server instruction set/IPaddress/port number/memory layout, database table names/column names, web server instruction set/memory layout, and local files used by the servers. There are other aspects of such a service that could also be diversified while not directly affecting the service functionality. Each method of diversification would have its own side effects and performance implications. There may also be other parts not identified that could also be used for diversification and coming up with a complete list is a difficult problem. The number and type of things that can be diversified will depend on the desired service and the software being used to provide that service. Another aspect of this technique is how often the randomization happens. In the case of a web service, it can be setup so that each user instance has a different randomization plan. It could also be implemented that each user request causes a new randomization of some of the methods.

Entities Protected: This technique aims to protect a web server.

Deployment: This would be deployed on a server.

Execution Overhead:

- This will vary with the number and type of diversification techniques implemented. Low-level emulated instruction randomization techniques would have a much higher overhead than randomizing the table names in a database. The overhead may be significant.

Memory Overhead:

- This will also depend on the diversification techniques implemented. If memory layout randomization is enabled, this could impose some overhead depending how it is implemented.

Network Overhead:

- Depending on the transformations applied to network protocols, this could increase the size of network traffic or increase the processing time of the traffic.

Hardware Costs:

- None

Modification Costs:

☒ Data

☒ Source Code

☒ Compiler/Linker

☒ Operating System

☐ Hardware

☐ Infrastructure

Expertise Required to Implement:

☐ Simple Configuration/Installer

☐ Complex Configuration (System Admin)

☐ Custom Programmer (General Knowledge)

☒ Custom Programmer (Expert/Low-Level/Kernel)

Expertise Required to Operate:

☒ Seamless

☐ Simple Configuration

☐ Complex Configuration (System Admin)

☐ Expert Operator

Kill Chain Phases:

☐ Reconnaissance

☐ Access

☒ Exploit Development

☒ Attack Launch

☐ Persistence

Interdependencies: Not all combinations may be desirable. Combining methods of randomization could affect the application in undesirable or unexpected ways. Certain combinations could also result in a large overhead.

Weaknesses: Weaknesses associated with this technique will vary depending on the randomization techniques implemented. Each technique will have its own weaknesses associated with it and combining techniques could introduce additional weaknesses not present in the independent methods. Some randomization methods may be limited by factors in the system such as the architecture. Despite all randomization, a higher level protocol may be vulnerable to attacks.

Types of Weaknesses:

☒ Overcome Movement ☒ Predict Movement ☒ Limit Movement ☒ Disable Movement

Impact on Attackers: If that attacker can leverage vulnerabilities in a service that allows control of the flow of the program, the attacker could still leverage more advanced techniques that do not rely on code injection. Implementing many of these methods will increase the amount of work an attacker has to do to exploit the system.

Availability: This was a research idea by the authors and did not appear to be implemented or available.

Additional Considerations: The report lacks many specifics. It is only applied to a web server. The actual impact of randomization in unknown. The overhead can be very large. Modifying the code to support all these additional randomization abilities could introduce additional bugs/vulnerabilities. Modifying the code to support all these additional randomization abilities could increase the maintenance complexity of application**.** Determining which components and subcomponents of an application or service could be a difficult and time-consuming task. The proposed randomization methods do not fix security vulnerabilities or other logic errors that are part of the design of the software. A similar technique is proposed in [59].

Proposed Research: This technique proposes many possible ways a specific web service could be randomized. Coming up with a method to identify and test these methods is not an easy task. Any part that is overlooked could become a potential attack vector. Determining which of these methods can be safely combined could also be another difficult task. It could be the case that combining two methods result in something breaking elsewhere in the service or system. Certain techniques will also have varying impacts on performance and the combination of different methods could cause unexpected performance issues. Overall, the composition of different randomization and diversification methods would need to be further researched for this technique to be more feasible.

Funding: Unknown

This page intentionally left blank.

REFERENCES

1. http://cybersecurity.nitrd.gov/page/moving-target.
2. C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," *Computer Security Applications Conference* (2006).
3. E.D. Berger and B.G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York, NY (2006) 158–168.
4. V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan, "Preventing Overflow Attacks by Memory Randomization," In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering* (2010).
5. M. Chew and D. Song, Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Dept. of Computer Science, Carnegie Mellon Univ., 2002.
6. "Windows ISV Software Security Defenses," Msdn.microsoft.com. Retrieved 10 April 2012.
7. "Mac OS X–Security–Keeps safe from viruses and malware," Apple.com. Retrieved 10 April 2012.
8. "The NX Bit and ASLR," Tom's Hardware, 25 March 2009.
9. "Pwn2Own day 2: iPhone, BlackBerry beaten; Chrome, Firefox no-shows," Ars Technica, 11 March 2011.
10. G. Zhu and A. Tyagi, "Protection against indirect overflow attacks on pointers," In *Proceedings of the Second IEEE International Information Assurance Workshop* (8–9 April 2004), pp. 97–106.
11. K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," In *Proceedings of the 26th Annual Computer Security Applications Conference*, New York, NY (2010), pp. 49–58.
12. W. Hu, J. Hiser, D. Williams, A. Filipi, J.W. Davidson, D. Evans, J.C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software," In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (2006), pp. 2–12.
13. X. Jiang, H.J. Wangz, D. Xu, and Y.-M. Wang, "RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization," In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems* (2007).
14. E.G. Barrantes, D.H. Ackley, S. Forrest, and D. Stefanovi, "Randomized instruction set emulation," *ACM Trans. Inf. Syst. Secur.* 8, 1 (February 2005).

15. S. Boyd and A. Keromytis, "SQLRand: Preventing SQL Injection Attacks," In *Applied Cryptography and Network Security*, vol. 3089, M. Jakobsson, M. Yung, and J. Zhou, Eds., Springer Berlin/Heidelberg (2004), pp. 292–302.

16. Z. Liang, B. Liang, L. Li, W. Chen, Q. Kang, and Y. Gu, "Against Code Injection with System Call Randomization," In *International Conference on Networks Security, Wireless Communications and Trusted Computing* (2009), vol. 1, pp. 584–587.

17. A.J. O'Donnell and H. Sethu, "On achieving software diversity for improved network security using distributed coloring algorithms," In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, New York, NY (2004), pp. 121–131.

18. T. Fraser, M. Petkac, and L. Badger, "Security Agility for Dynamic Execution Environments," Sep. 2002.

19. T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 4:1–4:54, Jul. 2010.

20. D. Chang, S. Hines, P. West, G. Tyson, and D. Whalley, "Program differentiation," In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, New York, NY (2010), pp. 9:1–9:8.

21. B. Salamat, A. Gal, and M. Franz, "Reverse stack execution in a multi-variant execution environment," In *Workshop on Compiler and Architectural Techniques for Application Reliability and Security* (2008).

22. D. Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic approaches to thwart adversary intelligence gathering," In *Proceedings of DARPA Information Survivability Conference & Exposition II* (2001), vol.1, pp. 176–185, doi: 10.1109/DISCEX.2001.932214.

23. C.M. Price, E. Stanton, E.J. Lee, J.T. Michalski, K.S. Chua, Y.H. Wong, and C.P. Tan, "Network Security Mechanisms Utilizing Dynamic Network Address Translation LDRD Project," Sandia National Labs, 2002 Nov 01.

24. J. Li, P.L. Reiher, and G.J. Popek, "Resilient self-organizing overlay networks for security update delivery," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, 189–202 (Jan 2004).

25. H. Moniz, N.F. Neves, M. Correia, and P. Verissimo, "Randomized Intrusion-Tolerant Asynchronous Services," *International Conference on Dependable Systems and Networks* (25–28 June 2006), pp. 568–577.

26. S. Antonatos, P. Akritidis, E.P. Markatos, and K.G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Comput. Netw.* 51, 12 (August 2007), 3471–3490.

27. E. Al-Shaer, "Toward Network Configuration Randomization for Moving Target Defense," *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats,* S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (eds.), New York: Springer, 2011, 153–159.

28. J.D Touch, G.G. Finn, Y.-S. Wang, and L. Eggert, "DynaBone: dynamic defense using multi-layer Internet overlays," In *Proceedings of DARPA Information Survivability Conference and Exposition,* vol. 2 (22–24 April 2003), pp. 271–276.

29. AFRL resources; personal communication.

30. M. Petkac and L. Badger, "Security agility in response to intrusion detection," *16th Annual Conference on Computer Security Applications,* (Dec 2000) pp. 11–20, doi: 10.1109/ACSAC.2000.898853.

31. D. Williams, W. Hu; J.W. Davidson, J.D. Hiser, J.C. Knight, and A. Nguyen-Tuong, "Security through Diversity: Leveraging Virtual Machine Technology," *IEEE Security & Privacy,* vol. 7, no.1 (Jan–Feb 2009), pp. 26–33.

32. B. Salamat, T. Jackson, G. Wagner, C. Wimmer, and M. Franz, "Runtime Defense against Code Injection Attacks Using Replicated Execution," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 4 (July–Aug 2011), pp. 588–601.

33. D.A. Holland, A.T. Lim, and M.I. Seltzer. "An architecture a day keeps the hacker away," *SIGARCH Comput. Archit. News* 33, 1 (March 2005), 34–41.

34. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: a secretless framework for security through diversity," In *Proceedings of the 15th conference on USENIX Security Symposium—Volume 15,* USENIX Association, Berkeley, CA (2006).

35. H. Okhravi, A. Comella, E. Robinson, and J. Haines, "Creating a cyber moving target for critical infrastructure applications using platform diversity," *Elsevier International Journal of Critical Infrastructure Protection*, Volume 5, Issue 1, March 2012, Pages 30–39, ISSN 1874-5482.

36. B.J. Min and J.S. Choi, "An approach to intrusion tolerance for mission-critical services using adaptability and diverse replication," *Future Gener. Comput. Syst.* 20, 2 (February 2004), 303–313. DOI=10.1016/S0167-739X(03)00146-8.

37. A. Saidane, V. Nicomette, and Y. Deswarte, "The Design of a Generic Intrusion-Tolerant Architecture for Web Servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, (Jan–March 2009), pp. 45–58.

38. A.K. Bangalore and A.K. Sood, "Securing Web Servers Using Self Cleansing Intrusion Tolerance (SCIT)," *Second International Conference on Dependability, 2009* (18–23 June 2009) pp. 60–65.

39. M. Crouse and E.W. Fulp, "A moving target environment for computer configurations using Genetic Algorithms," *4th Symposium on Configuration Analytics and Automation (*Oct. 31 2011–Nov. 1 2011), pp. 1–7.

40. Y. Huang and A. Ghosh, "Introducing Diversity and Uncertainty to Create Moving Attack Surfaces for Web Services," *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*,

S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (eds.), New York: Springer, 2011, 131–151.

41. Software Protection Initiative http://www.spi.dod.mil/lipose.htm.

42. P.E. Ammann and J.C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, Apr 1988 doi: 10.1109/12.2185.

43. A. Nguyen-Tuong, D. Evans, J.C. Knight, B. Cox, and J.W. Davidson, "Security through redundant data diversity," In *Proceedings of The 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2008,* Anchorage, Alaska, (2008), pp. 187–196, IEEE Computer Society.

44. C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Microsoft Research Tech. rep. (2008) MSR-TR-2008-120.

45. M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin, "End-to-End Software Diversification of Internet Services," *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*, S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (eds.), New York: Springer, 2011, 117–130.

46. M. Howard, "Fending Off Future Attacks by Reducing Attack Surface." Microsoft Corp, February 4, 2003.

47. Y. Huang, D. Arsenault, and A. Sood, "Incorruptible system self-cleansing for intrusion tolerance," *25th IEEE International Performance, Computing, and Communications Conference* (10–12 April 2006), pp. 4–496.

48. K. Scott and J. Davidson, "Safe virtual execution using software dynamic translation," *Proceedings of the 18th Annual Computer Security Applications Conference* (2002), pp. 209–218.

49. T. Wei, T. Wang, L. Duan, and J. Luo, "INSeRT: Protect Dynamic Code Generation against spraying," *International Conference on Information Science and Technology* (26–28 March 2011), pp. 323–328.

50. T. Jackson, B. Salamat, G. Wagner, C. Wimmer, and M. Franz, "On the effectiveness of multi-variant program execution for vulnerability detection and prevention," In *Proceedings of the 6th International Workshop on Security Measurements and Metrics* (2010), ACM, New York, NY, Article 7, 8 pages.

51. M. Hafiz and R.E. Johnson, "Security-oriented program transformations," In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies* (2009), F. Sheldon, G. Peterson, A. Krings, R. Abercrombie, and A. Mili (eds.). ACM, New York, NY, Article 12, 4 pages.

52. T. Jackson, C. Wimmer, and M. Franz, "Multi-variant program execution for vulnerability detection and analysis," In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information*

Intelligence Research (2010), F.T. Sheldon, S. Prowell, R.K. Abercrombie, and A. Krings (Eds.) ACM, New York, NY, Article 38, 4 pages.

53. B. Salamat, T. Jackson, A. Gal, and M. Franz, "Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space," In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009) ACM, New York, NY, 33–46, DOI=10.1145/1519065.1519071 http://doi.acm.org/10.1145/1519065.1519071.

54. M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," In *Proceedings of the 2010 Workshop on New Security Paradigms* (2010), ACM, New York, NY, 7–16, DOI=10.1145/1900546.1900550.

55. A. Nguyen-Tuong, A. Wang, J.D. Hiser, J.C. Knight, and J.W. Davidson, "On the effectiveness of the metamorphic shield," In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume* (2010), C.E. Cuesta (Ed.). ACM, New York, NY, 170–174.

56. Y. Weiss and E.G. Barrantes, "Known/Chosen Key Attacks against Software Instruction Set Randomization," In *Proceedings of the 22nd Annual Computer Security Applications Conference* (2006), IEEE Computer Society, Washington, DC, 349–360. DOI=10.1109/ACSAC.2006.33 http://dx.doi.org/10.1109/ACSAC.2006.33.

57. S.W. Boyd, G.S. Kc, M.E. Locasto, A.D. Keromytis, and V. Prevelakis, "On the General Applicability of Instruction-Set Randomization," *IEEE Trans. Dependable Secur. Comput.* 7, 3 (July 2010), 255–270.

58. A.D. Keromytis, "Randomized Instruction Sets and Runtime Environments Past Research and Future Directions," *IEEE Security and Privacy* 7, 1 (January 2009), 18–25.

59. F. Majorczyk and J.-C. Demay, "Automated Instruction-Set Randomization for Web Applications in Diversified Redundant Systems," *International Conference on Availability, Reliability and Security* (16–19 March 2009), pp. 978–983.

60. M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr, "Enhancing server availability and security through failure-oblivious computing," In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation—Volume 6*, USENIX Association, Berkeley, CA (2004), 21-21.

61. G. Portokalidis and A.D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," In *Proceedings of the 26th Annual Computer Security Applications Conference* ACM, New York, NY (2010), 41–48.

62. G.S. Kc, A.D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," In *Proceedings of the 10th ACM Conference on Computer and Communications Security,* ACM, New York, NY (2003), 272–280. DOI=10.1145/948109.948146 http://doi.acm.org/10.1145/948109.948146.

63. E.G. Barrantes, D.H. Ackley, T.S. Palmer, D. Stefanovic, and D.D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," In *Proceedings of the 10th ACM Conference on Computer and Communications Security,* ACM, New York, NY (2003), 281–289.

64. L.Q. Nguyen, T. Demir, J. Rowe, F. Hsu, and K. Levitt, "A framework for diversifying windows native APIs to tolerate code injection attacks," In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security,* R. Deng and P. Samarati (Eds.). ACM, New York, NY (2007), 392–394.

65. A.N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? the effectiveness of instruction set randomization," In *Proceedings of the 14th Conference on USENIX Security Symposium—Volume 14,* USENIX Association, Berkeley, CA (2005), 10-10.

66. V. Stankovic, A.N. Bessani, A. Daidone, I. Gashi, R.R. Obelheiro, and P. Sousa, "Enhancing Fault/Intrusion Tolerance through Design and Configuration Diversity," Paper presented at the *3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*, Jun 2009, Estoril, Lisbon, Portugal.

67. S.B.E. Raj and G. Varghese, "Analysis of intrusion-tolerant architectures for Web Servers," *International Conference on Emerging Trends in Electrical and Computer Technology*, (23–24 March 2011), pp. 998–1003.

68. C. Taylor and J. Alves-Foss, "Diversity as a computer defense mechanism," In *Proceedings of the 2005 Workshop on New Security Paradigms,* ACM, New York, NY (2005), 11–14.

69. R.A. Maxion, "Use of diversity as a defense mechanism," In *Proceedings of the 2005 Workshop on New Security Paradigms,* ACM, New York, NY (2005), 21–22. DOI=10.1145/1146269.1146277 http://doi.acm.org/10.1145/1146269.1146277.

70. K. Beznosov and P. Kruchten, "Towards agile security assurance," In *Proceedings of the 2004 Workshop on New Security Paradigms,* ACM, New York, NY (2004), 47–54.

71. P. Paruchuri, M. Tambe, F. Ord, and S. Kraus, "Security in multiagent systems by policy randomization," In *Proceedings of The Fifth International Joint Conference on Autonomous Agents and Multiagent Systems,* ACM, New York, NY (2006), 273–280.

72. J. Xu, Z. Kalbarczyk, and R.K. Iyer, "Transparent runtime randomization for security," In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems* (6–18 Oct. 2003), pp. 260–269.

73. L. Li, J.E. Just, and R. Sekar, "Address-Space Randomization for Windows Systems," In *Proceedings of the 22nd Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC, (2006), 329–338.

74. G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically Returning to Randomized lib(c)," In *Proceedings of the 2009 Annual Computer Security Applications Conference*, IEEE Computer Society, Washington, DC (2009), 60–69.

75. M. Abadi and G. Plotkin, "On Protection by Layout Randomization," In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, IEEE Computer Society, Washington, DC (2010), 337–351, DOI=10.1109/CSF.2010.30 http://dx.doi.org/10.1109/CSF.2010.30.

76. O. Whitehouse. "An analysis of Address Space Layout Randomization on Windows Vista," Symantec Advanced Threat Research, http://www.symantec.com/avcenter/reference/ Address_Space_Layout_Randomization.pdf

77. A. Sotirov and M. Dowd, "Bypassing browser memory protections in Windows Vista," http://www.phreedom.org/research/bypassing-browser-memory-protections/bypassing-browser-memory-protections.pdf

78. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ACM, New York, NY (2004), 298–307.

79. A. Sotirov and M. Dowd, "Bypassing Browser Memory Protections Setting back browser security by 10 years," http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf.

80. S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," In *Proceedings of the 14th Conference on USENIX Security Symposium—Volume 14*, USENIX Association, Berkeley, CA (2005), 17-17.

81. H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ACM, New York, NY (2007), 552–561.

82. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ACM, New York, NY (2010), 559–572.

83. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," In *Proceedings of the Second European Workshop on System Security*, ACM, New York, NY (2009), 1–8, DOI=10.1145/1519144.1519145 http://doi.acm.org/ 10.1145/1519144.1519145.

84. H. Xu and S.J. Chapin, "Improving address space randomization with a dynamic offset randomization technique," In *Proceedings of the 2006 ACM Symposium on Applied Computing*, ACM, New York, NY (2006), 384–391.

85. G. Novark and E.D. Berger, "DieHarder: securing the heap," In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ACM, New York, NY (2010), 573–584. DOI=10.1145/1866307.1866371 http://doi.acm.org/10.1145/1866307.1866371.

86. S. Bhatkar, D.C. DuVarney, and R. Sekar, "Address obfuscation: an efficient approach to combat a board range of memory error exploits," In *Proceedings of the 12th Conference on USENIX Security Symposium—Volume 12*, USENIX Association, Berkeley, CA (2003), 8-8.

87. Tyler Durden, “Bypassing PaX ASLR protection.” Phrack, Volume 0x0b, Issue 0x3b, Phile #0x09 of 0x12. http://www.phrack.org/issues.html?issue=59&id=9.

88. S. Forrest, A. Somayaji, and D. Ackley, “Building Diverse Computer Systems,” In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI),* IEEE Computer Society, Washington, DC (1997).

89. A. Smirnov and T. Chiueh, “A Portable Implementation Framework for Intrusion-Resilient Database Management Systems,” In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, IEEE Computer Society, Washington, DC (2004), 443.

90. D. Arsenault, A. Sood, and Y. Huang, “Secure, Resilient Computing Clusters: Self-Cleansing Intrusion Tolerance with Hardware Enforced Security (SCIT/HES),” In *Proceedings of The Second International Conference on Availability, Reliability and Security*, IEEE Computer Society, Washington, DC (2007), 343–350.

91. P. Sousa, A. Neves Bessani, M. Correia, N. Ferreira Neves, and P. Verissimo, “Resilient Intrusion Tolerance through Proactive and Reactive Recovery,” In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, IEEE Computer Society, Washington, DC (2007), 373–380.

92. Y. Huang and A.K. Ghosh, “Automating Intrusion Response via Virtualization for Realizing Uninterruptible Web Services,” *Eighth IEEE International Symposium on Network Computing and Applications* (9–11 July 2009), pp. 114–117.

93. S. Sidiroglou, O. Laadan, A.D. Keromytis, and J. Nieh, “Using Rescue Points to Navigate Software Recovery,” In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, IEEE Computer Society, Washington, DC (2007), 273–280, DOI=10.1109/SP.2007.38.

94. P.V. Prahbu, Y. Song and S.J. Stolfo, “Smashing the Stack with Hydra: The Many Heads of Advanced Polymorphic Shellcode,” Technical Report CUCS-037-09, Columbia University, August 2009.

95. Y. Song, M.E. Locasto, A. Stavrou, A.D. Keromytis, and S.J. Stolfo, “On the infeasibility of modeling polymorphic shellcode,” In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007), ACM, New York, NY, 541–551.

96. M. Frantzen and M. Shuey, “StackGhost: Hardware facilitated stack protection,” In *Proceedings of the 10th Conference on USENIX Security Symposium—Volume 10,* USENIX Association, Berkeley, CA (2001), 5-5.

97. C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks,” In *Proceedings of the 7th Conference on USENIX Security Symposium—Volume 7*, USENIX Association, Berkeley, CA (1998), 5-5.

98. C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointguardTM: protecting pointers from buffer overflow vulnerabilities," In *Proceedings of the 12th Conference on USENIX Security Symposium—Volume 12,* USENIX Association, Berkeley, CA (2003), 7-7.

99. V.R. Vasisht and H.-H.S. Lee, "SHARK: Architectural support for autonomic protection against stealth by rootkit exploits," In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, Washington, DC (2008), 106–116.

100. N. Joukov, A. Kashyap, G. Sivathanu, and E. Zadok, "An electric fence for kernel buffers," In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability,* ACM, New York, NY (2005), 37–43.

101. L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: a detection tool to defend against return-oriented programming attacks," In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security,* ACM, New York, NY (2011), 40–51.

102. V. Kiriansky, D. Bruening, and S.P. Amarasinghe, "Secure Execution via Program Shepherding," In *Proceedings of the 11th USENIX Security Symposium*, Dan Boneh (Ed.), USENIX Association, Berkeley, CA (2002), 191–206.

103. P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing Memory Error Exploits with WIT," In *Proceedings of the 2008 IEEE Symposium on Security and Privacy,* IEEE Computer Society, Washington, DC (2008), 263–277.

104. Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G.C. Necula, "XFI: software guards for system address spaces," In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation,* USENIX Association, Berkeley, CA (2006), 75–88.

105. G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.* (2005) 27, 3.

106. Common Attack Pattern Enumeration and Classification (CAPEC) http://capec.mitre.org/.

107. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *IEEE Transactions on Software Engineering (Jan/Feb 2012)*, IEEE Computer Society, Washington, DC (2012), vol. 38, no. 1.

108. Z. Wong and R.B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," *International Symposium on Computer Architecture*, ACM, San Diego, CA (2007), 494–505.

This page intentionally left blank.

LIST OF ACRONYMS

AES	Advanced Encryption Standard
API	application programming interface
ARCSYNE	Active Repositioning in Cyberspace for SYNchronized Evasion
ARP	Address Resolution Protocol
ASLR	address space layout randomization
CAPEC	Common Attack Pattern Enumeration and Classification
CPU	central processing unit
CSD	calling sequence diversity
DARPA	Defense Advanced Research Projects Agency
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
DOM	Document Object Model
DoS	denial of service
DynaBone	Dynamic Backbone
DYNAT	dynamic network address translation
ECB	Electronic Code Book
ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GID	group identification
GOTS	government off-the-shelf
HTML	Hypertext Markup Language

HTTP	Hypertext Transfer Protocol
I/O	input/output
IDS	intrusion detection system
IP	Internet Protocol
IPSec	IP Security
IRF	Instruction Register File
ISR	Instruction Set Randomization
LAN	local area network
MAC	media access control
MPLS	Multi-Protocol Label Switching
MUTE	Mutable Network
NASR	Network Address Space Randomization
NAT	Network Address Translation
NOP	no operation performed
NPV	Node Path Vector
OS	operating system
P2P	peer-to-peer
PPV	Parent Path Vector
REST	representational state transfer
RISE	randomized instruction set emulation
RITAS	randomized intrusion-tolerant asynchronous services
ROP	return-oriented programming
SCIT	self-cleansing intrusion tolerance

SQL	Structured Query Language
SSL	Secure Sockets Layer
TALENT	Trusted dynAmic Logical hEterogeNeity sysTem
TCP	Transmission Control Protocol
ToS	type of service
UDP	User Datagram Protocol
UID	user identification
VLAN	virtual local area network
VM	virtual machine
VPN	virtual private network
XSS	cross-site scripting